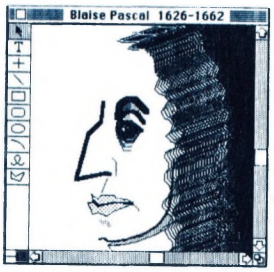
Programmieren
mit
PASCAL

aus: *ComputerKurs*-Magazin, Jahrgänge 1984/1985

Disziplin ist alles

Mit diesem Artikel beginnt eine Serie über PASCAL. Wir werden uns mit den Gedanken und Prinzipien, die zur Entwicklung führten, beschäftigen. Oft als „streng diszipliniert" bezeichnet, wurde PASCAL entworfen, um gute Programmier-Techniken zu fördern.



Als Mathematiker, Wissenschaftler und Philosoph entwickelte Blaise Pascal (1623–1662) im Jahre 1642 die erste mechanische Rechenmaschine. Die Programmiersprache PASCAL wurde zu Ehren dieses Beitrags zur Computerwissenschaft nach ihm benannt.

Die Programmiersprache PASCAL wurde im Jahre 1970 von Professor Niklaus Wirth in Zürich entwickelt. Die Sprache erhielt den Namen eines französischen Mathematikers und Philosophen des 17. Jahrhunderts – Blaise Pascal –, der die erste mechanische Rechenmaschine mit den vier Grundfunktionen erfunden hatte. Die Sprache PASCAL wurde durch ALGOL 60 stark beeinflußt und war eine direkte Antwort von Wirth auf die „überzüchtete" und komplexe Version ALGOL 68. Er verkündete, PASCAL würde:

- Programmstrukturen und Konzepte präzise auszudrücken erlauben;
- demonstrieren, daß eine kleine, maschinenunabhängige Sprache mit einem flexiblen Satz an Daten, Anweisungen und Programmstrukturierungs-Details als allgemeines Problemlösungswerkzeug verwendet werden kann;
- bei der Organisation von großen Programmen und komplexen Software-Projekten größere Sicherheit bieten;
- über umfassende Fehlerprüfroutinen verfügen (speziell während der Compilierung) und dadurch ideal als Lehrsprache für Anfänger verwendbar sein;
- sinnvoll auf Microcomputern eingesetzt werden können.

Alle diese Vorsätze sind mit großem Erfolg realisiert worden. Ein kleiner PASCAL-Compiler belegt in der Regel etwa 25 KByte. Obwohl PASCAL nur über ein kleines Vokabular verfügt, das leicht erlernbar ist, ist es trotzdem viel leistungsstärker als manche andere Computersprache. Viel wichtiger ist jedoch, daß PASCAL im Ausdruck erheblich stärker ist. Dies gilt zum einen für die Art, in der Algorithmen geschrieben werden können, und zum anderen für die Einfachheit, mit der Daten umschrieben werden können, egal, wie kompliziert sie sind.

Vielleicht sogar der größte Vorzug von PASCAL ist die eben erwähnte einfache Ausdrucksweise für Algorithmen. Sie können so programmiert werden, wie man über sie nachdenkt. Diese Freiheit beim Programmieren macht PASCAL zu einem idealen Problemlösungswerkzeug und nicht zu einem weiteren Bestandteil des betreffenden Problems. Die Sprache hat viele weitere Vorteile. PASCAL ist eine Compilersprache, was nicht nur bedeutet, daß die Programmausführung viel schneller wird, sondern auch, daß kein Speicherplatz unnötig mit Interpretern und Programmtexten belegt wird. – Alles was Sie brauchen, ist der compilierte „Object"-Code.

In der Tat ist es gerade bei langen Programmen viel einfacher, in PASCAL statt in BASIC zu programmieren. PASCAL läßt eine „schlampige" Programmierung gar nicht erst zu. Betrachtet man die Sache etwas näher, so ist die aufgezwungene Disziplin nicht strenger als die, die zur effektiven und sicheren Organisation eines Programms ohnehin notwendig ist. Die eingesparte Zeit beim späteren „Debuggen" eines Programms wiegt dies bei weitem auf. Ferner bedeutet dies, daß ein Grundwissen über PASCAL ein enormer Vorteil für das Verständnis der Sprachen der nächsten Dekade ist – beispielsweise bei Modula-2, OCCAM oder ADA.

Ein Programmierer, der von BASIC auf PASCAL umsteigen will, wird bei einem ersten Blick auf ein PASCAL-Programm den krassesten Unterschied in den sehr umfangreichen und eigenartig aussehenden Definitionen (die manchmal fast sinnlos erscheinen) sehen. Der gesamte Anfang eines längeren PASCAL-Programms scheint absolut nichts zu „tun". Dies liegt zum Teil daran, daß Sie eigene Worte innerhalb der Programmiersprache verwenden können. Eine entsprechende Definition muß selbstverständlich vor dem eigentlichen Programmstart erfolgen, damit PASCAL die Worte als Befehle interpretieren kann. PASCAL ermöglicht Ihnen so, am Anfang eines Programms neue Befehle festzulegen, wie ClearScreen oder Pause (Anzahl der Sekunden), und diese dann im eigentlichen Programm zu verwenden. Das sieht dann so aus:

```
begin
  ClearScreen;
  Write ("Hallo!");
  Pause (3);
  . . . usw.
```

Überraschend einfach

PASCAL überrascht den Programmierer mit einer ausgezeichneten Betrachtung des Konzepts eines Computersystems, so daß Daten und Berechnungen in einer natürlichen, logischen Form definiert und ausgedrückt werden können.

Die wesentlichen Details von Pascal

- Vollständig freies Format und freie Gestaltung des Programmtextes
- Flexible Bedingungen bei der Benennung von Programmobjekten
- Die Möglichkeit zur Definition neuer Befehlswörter
- Einfache und einheitliche Syntax
- Modulare Programmstruktur
- Flexible Kontrolle von Daten und Berechnungen
- Sehr schnelle Compilierung sowie Fehler-Diagnose
- Ein kleiner, hocheffizienter Compiler

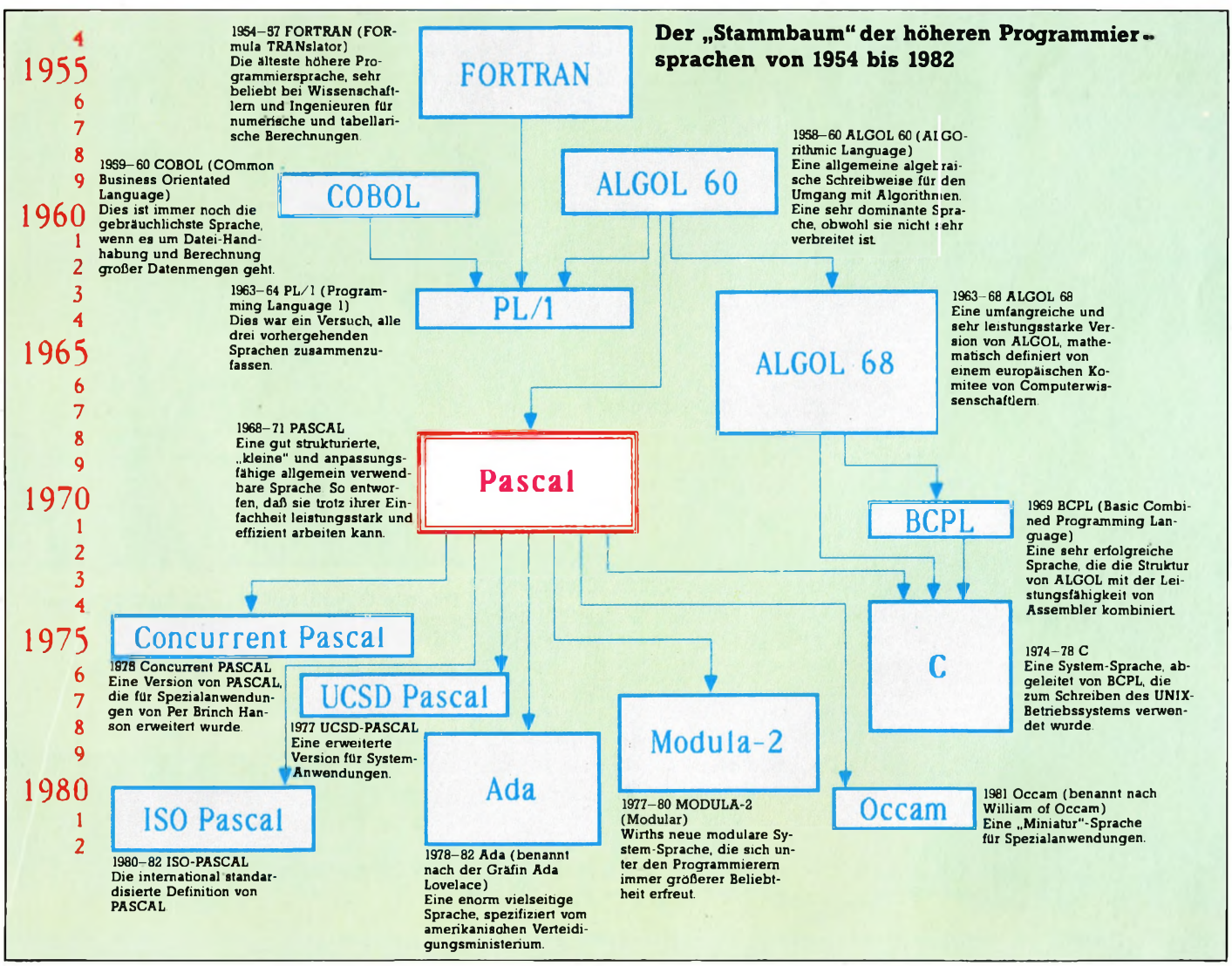
Das Diagramm, das den „Stammbaum" höherer Programmiersprachen zeigt, beinhaltet nur die Haupteinflüsse der wichtigsten befehlsorientierten Compilersprachen. Daher werden funktionelle Sprachen, wie LISP oder PROLOG, nicht berücksichtigt. Auch FORTH wird nicht aufgeführt, da es kaum klassifizierbar ist. Der Hauptstamm beginnt mit ALGOL 60. Danach kommt kaum eine moderne Sprache, die nicht direkt oder indirekt von PASCAL beeinflußt wurde.

Neben dem Einsatz als Lehrsprache eignet sich PASCAL auch für kommerzielle und systemspezifische Anwendungen. PASCAL ist für die Entwicklung von Finanzpaketen und Compilern verwendet worden. Das „p-system", das an der UCSD (Universität von Kalifornien in San Diego) gegen Ende der siebziger Jahre entworfen wurde, ist sowohl in PASCAL entwikkelt als auch geschrieben worden. Die Software für Lisa und Macintosh von Apple wurde, einschließlich der Betriebssysteme, hauptsächlich in PASCAL bzw. der abgeleiteten Sprache CLASCAL geschrieben. Viele tausend professionelle Programmierer haben sich auf PASCAL umgestellt, entweder an den Universitäten auf DEC oder VAX oder eventuell mit Hilfe eines Apple II mit einer Version des „p-systems", bekannt als Apple-PASCAL.

Programmentwicklungen

Mitglieder des ursprünglichen UCSD-Teams findet man heute bei der Arbeit an MODULA-2-Compilern oder anderen Systemen sowie beim Schreiben von Anwenderprogrammen für viele Software-Häuser. – Und alle arbeiten mit PASCAL oder einer leicht abgewandelten Version dieser Sprache.

Ein beachtenswerter Punkt ist, daß all dies ohne die finanzielle Unterstützung kommerzieller Firmen geschah. PASCAL war allein durch seine Vorzüge so erfolgreich, daß es keiner verkaufsfördernden Werbung von Software-Produzenten bedurfte. Unsere Serie über PASCAL konzentriert sich auf die Standard-Fassung (ISO-PASCAL). Bei Themen wie Grafik oder Routinen behandeln wir auch einige maschinenspezifische Besonderheiten, die einzeln erläutert werden.



Wort für Wort

Im zweiten Teil unserer PASCAL-Serie befassen wir uns mit grundlegenden Voraussetzungen wie der Syntax und dem Vokabular.

Compiler
Eine komplette professionelle PASCAL-Ausrüstung kann oft mehr kosten als ein Heimcomputer. Doch gibt es inzwischen viele Compiler, die zu einem vernünftigen Preis erhältlich sind. Wir zeigen Ihnen hier eine Auswahl der zur Zeit erhältlichen Programmpakete.

Das erste Problem bei der Verwendung einer Compilersprache ist der Umgang mit dem mehrstufigen Arbeitsprozeß, selbst bei einem kleinen Programm. Als erstes muß der Source-Code mit Hilfe eines Editors oder eines Textverarbeitungsprogramms eingegeben werden. Dann, nachdem man den Source-Code auf Cassette oder Diskette gespeichert hat, muß der Compiler geladen und anschließend instruiert werden (oftmals in Form einer komplexen Befehlszeile), den Source-Code in Maschinen-Code umzuwandeln bzw. zu compilieren. Abschließend muß dann das „Object"-File einem Speicherbereich zugeordnet und mit den entsprechenden notwendigen „Run-Time-Library"-Routinen verbunden werden. In den meisten Fällen kann das Programm dann ohne weiteren Aufwand geladen und gestartet werden, doch sollte der Compiler einen sogenannten „Pseudo-Code", auch Zwischencode genannt, verwenden, muß ein Run-Time-Interpreter benutzt werden, um das Programm auszuführen.

Doch fast alle für Heimcomputer erhältlichen PASCAL-Versionen vermeiden diese Probleme weitestgehend. Bei den besten Versionen ist es sogar möglich, während der Programmentwicklung den Source-Code, den Compiler und das Object-Programm gleichzeitig im Speicher zu haben. Die Effizienz und der geringe Platzbedarf von PASCAL machen dies möglich. Lediglich beim Schreiben von Programmen muß man sich mit einem etwas komplizierteren Arbeitsablauf abfinden.

Jedes System verfügt über einen individuellen Satz an Befehlen zur Kontrolle des Editors und des Compilers. Oftmals ist nur ein einfaches E für Editieren, ein C für Compilieren und ein R für RUN (Programmstart) alles, was Sie wissen müssen. Zunächst befassen wir uns mit der korrekten Syntax, die für jedes Programm, egal wie einfach oder komplex es sein mag, eingegeben werden muß. Glücklicherweise ist PASCAL so gut standardisiert, daß es kaum Unterschiede zwischen den einzelnen Dialekten gibt (im Gegensatz zu den verschiedenen BASIC-Versionen). Lediglich gegen Ende des Kurses werden wir uns mit einigen wenigen Ausnahmen befassen. Lassen Sie uns nun unser erstes vollständiges PASCAL-Programm betrachten:

```
Program First (output);
Const
   Message='Pascal-Programmierung';
Begin
   write (Message)
End.
```

Geben Sie zunächst die Zeilen ein. Anschließend compilieren und starten Sie es. Sollten Sie irgendwelche Beanstandungen vom Compiler erhalten, lesen Sie die Fehlermeldung sorgsam durch, und versuchen Sie, den Fehler zu erkennen. Jedes Zeichen muß exakt eingegeben werden. Beachten Sie besonders das Semikolon am Ende der ersten und dritten Zeile sowie den Punkt am Ende des Programms. Wenn Sie das Programm korrekt eingegeben haben, werden Sie die folgende Meldung auf dem Bildschirm sehen:

```
Pascal-Programmierung
```

Das Programm mag etwas trivial erscheinen, doch es demonstriert die allgemeine Form, die jedem PASCAL-Modul (bzw. Programm, Unterroutine oder Funktion) zu eigen ist. Es gibt jeweils drei separate Teile:

1. Die Kopfzeile, in diesem Fall die Programm-Überschrift.
2. Deklarationen und Definitionen, in diesem Fall eine einzige konstante Definition.
3. Der „Körper", der alle ausführbaren Anweisungen enthält.

Die erforderliche Syntax von PASCAL, zumindest für die grundlegenden Funktionen der Sprache, kann am besten durch sogenannte „Syntax-Diagramme" definiert werden. Sie sind vergleichbar mit der Straßenkarte

Paket: Hisoft PASCAL für:
Spectrum, Schneider und MSX Computer

Beschreibung:
Eine vom Standard unbedeutend abweichende Version, aber trotzdem ein guter Kauf. Das Paket wird mit einem eigenen Editor und einer „Turtle Graphics Library" (Bibliothek) geliefert.

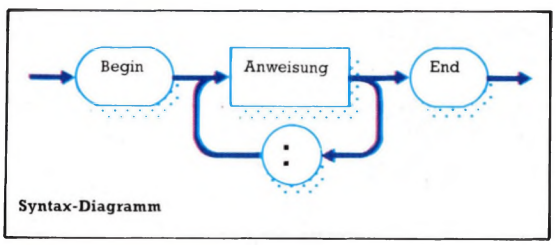
Paket: Acorn Iso PASCAL für:
Acorn B, Electron

Beschreibung:
Sehr hochwertig. Es gibt zwei Compiler: einer im ROM mit einem semi-intelligenten Editor und Compilierung im Speicher, der andere als diskettenorientiertes System für Besitzer eines Rechners mit 6502-CPU.

Paket: Oxford PASCAL für:
Commodore 64

Beschreibung:
Ein ebenfalls guter Kauf, produziert von Limbic Systems.

eines Einbahnstraßen-Systems. Ein legaler Weg durch das Diagramm führt von links oben nach rechts unten. Jeder Kasten, durch den wir auf diesem Weg kommen, ist entweder eine „syntaktische Einheit" (das bedeutet, sie repräsentiert sich selbst), dargestellt durch eine runde Box, oder ein anderes Objekt, das an einer anderen Stelle durch ein separates Syntax-Diagramm definiert wird. Diese werden in einem rechteckigen Kasten dargestellt.



Syntax-Diagramm

Bei der Betrachtung des Gesamtdiagramms eines Programms kann man sehen, daß die Worte „Begin" und „End" als Teile des PASCAL-Vokabulars definiert sind und somit keine weiteren Diagramme zu Erklärung ihrer Bedeutung benötigen. In der Tat haben bei PASCAL nur 35 Wörter eine festgelegte Bedeutung. Der Vollständigkeit halber haben wir sie alle am Ende dieses Kursteils aufgelistet. Unser erstes Programm verwendet nur vier dieser Wörter: Program, Const (die Abkürzung für constant = konstant), Begin und End. Das auf „Program" folgende Wort ist ein sogenannter „Identifikator", der den Programm-Namen angibt. Es kann aber auch ein anderes „legales" Wort verwendet werden.

Wenn man mit einem Buchstaben beginnt, und nur Buchstaben oder Zahlen verwendet, gibt es eine unbestimmbar große Anzahl von Wörtern, die Sie benutzen können. Trotzdem ist es nicht möglich, ein reserviertes Wort zu verwenden. Die folgenden Wörter sind beispielsweise alle erlaubt:

Name
PASCAL
ProgramEins
N
XYZ123
Adresse12
KnutRaber
EinsehrlangerIdentifikator

Die jetzt folgenden Wörter sind nicht legal:

Prog-1
ZEHN°
Feuerzeug.Ofen
and
Zeit$Ecke
Wiegeht's
1001Dalmatiner
Es regnet

Diese Wörter sind nicht erlaubt, da sie Zeichen enthalten, die nicht alphanumerisch sind, mit einer Zahl beginnen oder (im Fall and) eines der von PASCAL reservierten Wörter darstellen. Das letzte Beispiel ist illegal, da eine Leerstelle zum Trennen der Wörter verwendet wurde, wobei die Wörter an sich („Es" und „regnet") separat legal wären. Bei PASCAL besteht im allgemeinen, ähnlich wie in der englischen Sprache, kein Unterschied zwischen Klein- und Großbuchstaben, obwohl es einige Versionen gibt, bei denen reservierte Wörter in Großbuchstaben einzugeben sind.

Neben Leerstellen und dem Ende einer Zeile gibt es in der PASCAL-Syntax eine weitere Möglichkeit, die als Trennung verwendet werden kann – eine Anmerkung. Sie kann an einer beliebigen Stelle erscheinen, ausgenommen mitten in einem Wort. Anmerkungen werden durch geschwungene Klammern abgegrenzt.

Lassen Sie uns ein etwas komplexeres PASCAL-Programm betrachten:

```
Program ProgramZwei (input, output);
  {Quadrat einer Zahl}
Const
  Prompt = 'Gib Zahl ein:';
Var
    zahl : integer;
Begin
  WriteLn;
  WriteLn;
  write (Prompt);
  read (zahl);
  WriteLn (zahl, 'Quadrat ist', zahl*zahl)
End.
```

Beachten Sie, daß wir den Identifikator „input" in die Kopfzeile integriert haben. „Input" und „output" werden von PASCAL benötigt. Sie identifizieren die externen Dateien, mit denen das Programm kommuniziert. Bei einem Heimcomputer sind das normalerweise die Tastatur und der Bildschirm.

Der Speicher zum Ablegen dieser Parameter wird durch die Var-Deklaration reserviert, wobei dies in unserem Beispiel eine einzige ganze Zahl ist. Im Gegensatz zu BASIC, das normalerweise nur zwischen Zahlen und Strings unterscheiden kann (durch Anhängen eines Dollar-Zeichens an den Identifikator), verfügt PASCAL über einen nahezu unbeschränkten Bereich an verfügbaren Datentypen. Daher ist es wichtig, dem Compiler mitzuteilen, wieviel Speicherplatz er zum Speichern der jeweiligen Datensätze reservieren muß.

Der Cursor bleibt direkt an der Position nach Darstellung der Meldung stehen, so, als hätten wir eine BASIC-PRINT-Anweisung mit einem Semikolon verwendet. Dies ist kein Fehler, sondern ein Merkmal der write-Funktion. Wann immer man eine neue Zeile benötigt, muß man die alternative Anweisung WriteLn verwenden. Ln steht für Line (Zeile). Es ist recht sinnvoll, wenn man ein großes W und L verwendet, um den Anfang der zusammengesetzten Wörter besser zu kennzeichnen. Eine einfache WriteLn-Anweisung ohne weitere Parameter erzeugt eine neue Zeile.

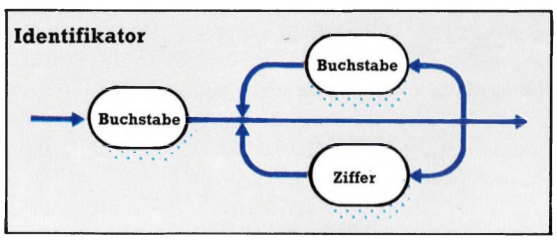
35 reservierte PASCAL-Wörter

And	Nil
Array	Not
Begin	Of
Case	Or
Const	Packed
Div	Procedure
Do	Program
Downto	Record
Else	Repeat
End	Set
File	Then
For	To
Function	Type
Goto	Until
If	Var
In	While
Label	With
Mod	

Datentypen

PASCAL verwendet vier einfache Datentypen innerhalb der Variablen-Deklarationen: Integer-, Real-, Zeichen- und Boolesche Variablen.

PASCAL bietet vier vordefinierte Datentypen, die als Integer, Real, Char und Boolean bezeichnet werden. Zahlen werden als reale Zahlen bezeichnet, wenn sie einen abtrennbaren Teil haben („Real"). Natürliche ganze Zahlen sind „Integer". Der tatsächlich verfügbare Bereich von Zahlen ist davon abhängig, wieviele Bytes zum Speichern des jeweiligen Datentyps verwendet werden.



Der Bereich der Integer-Zahlen wird bei Ihrem Compiler entweder von -32 768 bis 32 767 oder von -2.147.483.648 bis 2.147.483.647 reichen. Dies hängt davon ab, ob zum Speichern zwei oder vier Bytes verwendet werden. PASCAL bezeichnet den vorgegebenen Wert der höchsten Integer-Zahl als MaxInt. Somit läßt sich dieser Wert mit der folgenden Anweisung herausfinden:

```
WriteLn ('MaxInt ist: ',MaxInt)
```

Reale Zahlen können ebenfalls nur in einem bestimmten Bereich und mit einer bestimmten Genauigkeit gespeichert werden. Er reicht im allgemeinen von minus bis plus 1,7E38 mit einer Genauigkeit von mindestens sechs bis sieben Stellen nach dem Komma.

Beachten Sie, daß eine reale Zahl (für den Rechner) immer einen Dezimalpunkt haben muß (in normaler Schreibweise ein Dezimalkomma), der die ganze Zahl von den Dezimalstellen trennt. Beides muß eingegeben werden. Somit sind 0.1 und 1.0E-1 möglich, doch .1 oder 1E-1 sind illegal.

„Char" ist die Abkürzung für Character. Ein Wert dieses Datentyps ist ein Zeichen aus dem ASCII-Satz des Computers. PASCAL gewährleistet seine eigene Anpassungsfähigkeit, indem es folgende Punkte festlegt:

– Die Zeichen A bis Z werden alphabetisch geordnet. Das bedeutet, daß A einen kleineren Wert hat als B, B einen kleineren Wert als C und so weiter.

– Die Zahlen-Zeichen 0 bis 9 werden der bekannten Reihenfolge nach geordnet.

Jedes ASCII-Zeichen erhält einen numerischen Code, der ein Wert des untergeordneten Bereiches der Integer-Datentypen ist. Die ASCII-Codes sind in einem Bereich von 0 bis 127 definiert. Bei vielen Computern geht der Bereich sogar bis 225, wobei die zusätzlichen Codes spezielle Grafikzeichen repräsentieren. Man kann jeden gewünschten Zeichensatz problemlos auf der Skala der Ordnungszahlen darstellen, die vom Computer intern verwendet werden. PASCAL verfügt über die vordefinierte Funktion Ord: Sie gibt als Ergebnis den Integer-Wert des angegebenen Argumentes aus. Somit ist Ord(A) das Äquivalent des Wertes 65 im ASCII-Zeichensatz. Eine andere Funktion, chr, kehrt diesen Vorgang um – chr (65) ergibt das Zeichen A.

Der Bereich der „Zeichen"-(Char) und der Integer-Werte ist für jede PASCAL-Version fest definiert, das heißt, er basiert auf einer geordneten Skala bekannter Konstanten. Aus diesem Grund werden sie als „Ordnungs"- bzw. „Skalar"-Datentypen bezeichnet. Ganz gleich, welcher Wert vorliegt, man kennt immer den vorangegangenen und folgenden Wert. Diese angrenzenden Werte können durch die beiden folgenden Skalar-Funktionen ganz einfach ermittelt werden.

```
pred(item) (vorangegangener Wert)
succ(item) (nachfolgender Wert)
```

Somit ergibt succ(3) als Ergebnis immer den Zeichenwert 4, wogegen pred(Z) nur bei bestimmten Zeichensätzen (zum Beispiel ASCII) Y als Ergebnis ausgibt. Pred(MaxInt) ergibt entweder den Wert 32 766 oder 2.147.483.646. Die chr-Funktion kann nur mit einem Argument verwendet werden, das ein Zeichen-Code ist. Alle anderen Skalar-Funktionen können dagegen mit beliebigen Skalar-Datentypen verwendet werden.

Boolesche Variablen

„Boolesche Variablen" sind die einfachsten aller Skalar-Datentypen, da nur zwei Werte möglich sind – falsch und wahr (in dieser Reihenfolge). Die Skalar-Funktionen können daher auf jeden dieser Werte angewendet werden – der Ordnungs-Wert von falsch ist 0, und ord(wahr) ergibt 1.

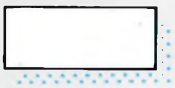
Der folgende Programmteil zeigt alle einfa-

Symbole

Die folgenden drei Symbole werden allgemein in PASCAL-Syntax-Diagrammen verwendet.

● Das ovale Symbol repräsentiert reservierte PASCAL-Worte oder Zeichen, die keiner weiteren Erklärung bedürfen (wie "Buchstabe" oder "Ziffer").

● Ein Kreis repräsentiert einen Pascal-Operator (+, -, *, . usw.).



● Das rechteckige Symbol steht für ein Wort oder einen Abschnitt, der sein eigenes separates Syntax-Diagramm hat.

chen Ordinal-Datentypen, die in einem PASCAL-Programmtext auftreten können.

```
CONST
  VAT          =0.15;
  spalten      =40;
  leerstellen  ='';
  fehlersuche  =falsch;
```

Mit dem Gleichheits-Zeichen werden die Identifikatoren bestimmten Werten zugeordnet. Der Doppelpunkt(:) trennt im VAR-Abschnitt neu deklarierte Variablen-Identifikatoren vom jeweiligen Datentyp. Betrachten Sie das folgende Beispiel:

```
VAR
  verhaeltnis  :real;
  zahl         :integer;
  symbol       :char;
  fertig       :boolean;
```

Der Zuordnungs-Operator

Wenn wir diesen Variablen Werte zuordnen wollen, wird der sogenannte zusammengesetze „Zuordnungs-Operator" (:=) verwendet. Dies ist für die eindeutige Unterscheidung zwischen den drei Arten von Operationen sehr hilfreich.

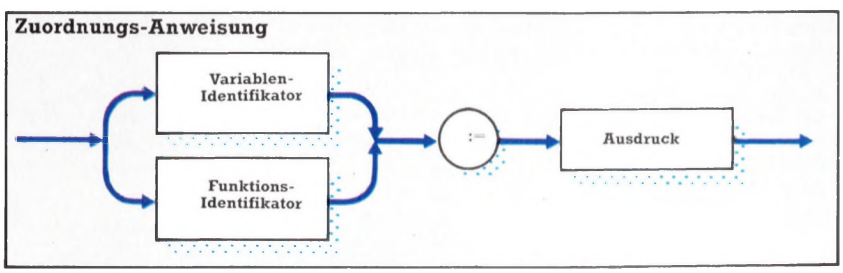
CONST-Definitionen ordnen permanente Werte zu. VAR-Deklarationen reservieren lediglich Speicherplatz. Durch Zuordnung erhält der Identifikator einen Wert.

Wenn zwei oder mehrere Anweisungen als Teil eines Arbeitsvorganges ausgeführt werden müssen, kann man sie als „zusammengesetzte Anweisung" (durch ein Semikolon getrennt) zwischen die Worte BEGIN und END schreiben. Sie haben sicherlich bereits zusammengesetzte Anweisungen gesehen; denn diese Form kommt in jedem umfangreicheren Programm vor. Im folgenden sehen Sie ein PASCAL-Programm, das viele der bereits besprochenen Details beinhaltet. Übrigens werden ab dieser Folge die reservierten Wörter in Großbuchstaben geschrieben, damit Sie sie besser von den Identifikatoren unterscheiden können.

```
PROGRAM Kreis (input, output);
CONST
  pi           =3.1415926536;
  meldung      ='Gib Radius ein:';

VAR
  radius,
  umfang       :real;

BEGIN
  WriteLn;
  write(meldung);
  read (radius);
  umfang := pi*radius*radius;
  WriteLn;
  WriteLn ('Der Umfang eines Kreises',
    'mit dem Radius', radius:8:3);
  WriteLn ('betraegt:', umfang:10:3)
END.
```

In diesem Beispiel gibt es zwei Aspekte, die beachtet werden sollten. Der erste ist, daß im VAR-Abschnitt zwei Identifikatoren desselben Datentypes deklariert werden – beide real. Wie Sie sehen, sind zwei separate Deklarationen nicht notwendig, da eine größere Anzahl von Namen in PASCAL einfach durch Kommata getrennt werden kann. Der zweite Aspekt ist die Art der Bildschirmausgabe. Sie wird verwendet, um die vorgegebene wissenschaftliche Schreibweise von realen Zahlen zu umgehen. Wie Sie sehen, kann man ganz nach Wunsch zwei Integer-Zahlen, getrennt durch Doppelpunkte, spezifizieren. Damit wird eine bestimmte Feldbreite zur Darstellung der gesamten Zahl und ihres abtrennbaren Teiles festgelegt.

In unserem Kreis-Programm sind sowohl für den Radius als auch für den Umfang drei Dezimalstellen vorgesehen. Da der Umfang eine größere Zahl sein wird, sind insgesamt zehn Zeichenpositionen vorgesehen, zwei mehr als für den Radius. Diese Integer-Werte müssen ein eventuelles Vorzeichen sowie die Darstellung des Dezimalpunktes vor dem abtrennbaren Teil zulassen.



PASCAL rundet automatisch die letzte Stelle, um die maximale Genauigkeit in einem bestimmten numerischen Bereich anzugeben. Außerdem kann jede Variable oder jeder Ausdruck verwendet werden, – nicht nur Konstanten. Bei allen anderen Datentypen wird nur ein Integer-Wert zur Bestimmung der Darstellungsform benötigt. Wenn man eine Feldbreite von eins festlegt, werden Integer-Werte ohne Leerstellen ausgegeben. Soll die Bildschirmausgabe tabellarisch gegliedert sein, müssen Sie daran denken, zusätzliche Leerstellen einzusetzen. Betrachten Sie das folgende Beispiel:

```
WriteLn ('Total:' :20,gewicht:1,' Kg.')
```

Achten Sie darauf, zum Trennen der einzelnen Ausgaben Leerstellen einzufügen. Wenn Sie beispielsweise die Zahlen 12 und 34 in dieser Form ausgeben lassen, so erhalten Sie auf dem Bildschirm 1234.

Option und Auswahl

In diesem PASCAL-Kurs werden die Strukturen der Vergleiche erklärt. Dabei behandeln wir die IF-Anweisung und die CASE-Anweisung, eine Struktur, die Mehrfach-Auswahlen ermöglicht.

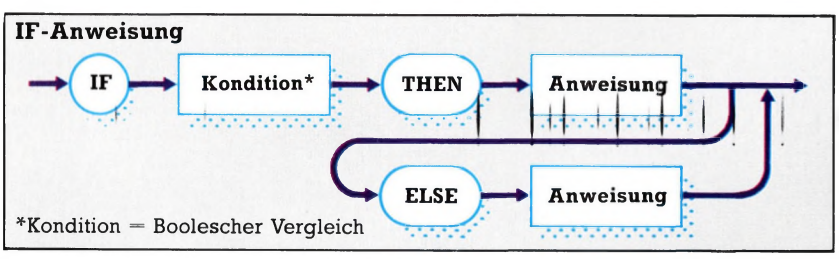
Die IF-Anweisung ist bei PASCAL ähnlich zu handhaben wie bei den meisten anderen Programmiersprachen. Durch Einrücken untergeordneter Begriffe wird die logische Struktur einer solchen IF-Anweisung erkennbar werden. Im folgenden sehen Sie zwei IF-Anweisungen:

```
IF zahl=grenze
   THEN
      WriteLn ('Kein Platz')
   ELSE
      write ('Naechste Eingabe?');
IF nummer > maximum THEN
   maximum : = nummer
```

Im zweiten Beispiel bedeutet das Fehlen von einer ELSE-Bedingung:

```
ELSE
   {mache nichts}
```

Das Schreiben der reservierten Wörter THEN und ELSE mit derselben Einrückung hilft Ihnen beim Nachvollziehen der Programmstruktur.



PASCAL war die erste Programmiersprache, die die Verwendung vorbestimmter Skalen einführte. Diese Methode ist sehr hilfreich, da sie einen umfassenden Überblick über die verschiedenen Daten-Typen in einem Programm ermöglicht, ohne daß man ständig Daten in numerische Codes umwandeln muß. Wir haben bereits gesehen, wie eine einfache Konstanten-Definition uns dabei helfen kann:

```
CONST
   breite =80;
```

Der Konstanten-Identifikator „breite" kann dann im gesamten Programm als Referenz in bezug auf die Anzahl der Spalten des Bildschirms oder des Druckers verwendet werden. Um das Programm für einen Drucker umzuschreiben, der nur eine Spaltenbreite von 40 hat, braucht man nur eine Zeile in den Definitionen am Anfang des Programms zu ändern. Mit dieser Änderung werden automatisch alle Berechnungen der Datenausgabe entsprechend angepaßt.

Wenn man Farbgrafiken verwendet, könnten die verfügbaren Farben durch eine entsprechende Skala von Zahlen dargestellt werden (beispielsweise rot = 1, gruen = 2). Doch dann besteht die Gefahr, daß man die Wurzel aus gruen ziehen oder blau mit gelb multiplizieren müßte! Dies ist nicht nur unlogisch, sondern auch eine potentielle Fehlerquelle. Der TYPE-Definitionsabschnitt eines PASCAL-Programms kann jedoch dazu verwendet werden, einen völlig neuen Skalar-Typ zu definieren. Hierzu braucht nur eine Liste von Identifikatoren, die alle konstanten Werte der Skala repräsentieren, bestimmt zu werden. Zum Beispiel:

```
TYPE
   farbe=(rot,gruen,gelb,blau,magenta,cyan);
```

Da es eine Definition und keine Deklaration ist, wird das Gleichheitszeichen zur Zuordnung der in Klammern befindlichen geordneten Werte zum Identifikator (farbe) verwendet. Die Identifikatoren werden intern durch Integer-Werte (mit der normalen Zählweise beginnend bei Null) repräsentiert. Diese numerische Repräsentation wird automatisch durch den Compiler organisiert und entspricht in etwa den Codes der Computer-Zeichensätze. Jeder Farbwert hat eine Ordnungs-Zahl, die man mit der Skalar-Funktion "ord" bestimmen kann. Im hier gezeigten Beispiel würde ord(rot) den Wert 0 und ord(cyan) den Wert 5 als Ergebnis ausgeben. Wir können nun eine Variable in der bereits bekannten Weise deklarieren:

```
VAR
   farbwert : farbe;
```

Hiermit wird ein Identifikator (farbwert) als Datenbestandteil des Datentyps "farbe" deklariert. Ähnlich verhält es sich mit der folgenden Deklaration:

```
VAR
   buchstabe : zeichen;
```

Diese Deklaration gibt die Zeichenart eines Datenobjektes mit dem Namen "buchstabe" an. Die einzigen Operationen, die für selbstbestimmte Datentypen definiert sind, sind Vergleiche und die Skalar-Funktionen. So könnte man beispielsweise schreiben:

```
IF farbwert < cyan THEN
   farbwert := succ(farbwert)
```

Zeichen und Zahlen können als Parameter in "write-" und "WriteLn"-Anweisungen verwendet werden, doch

```
WriteLn (farbwert)
```

wäre illegal. Wenn man die Namen ausdrukken will, muß man die Farbwerte in Form von Zeichen-Strings darstellen. Dies ist eine ideale Anwendung für die andere bereits erwähnte PASCAL-Struktur, die CASE-Anweisung.

Bis jetzt haben Sie gesehen, daß die Handhabung der IF-Anweisung in PASCAL der anderer Programmiersprachen emtspricht. Durch die freie Programmtexteingabe von PASCAL wird sie sogar noch einfacher nachvollziehbar. Trotzdem kann es vorkommen, daß man mehrere Entscheidungen auf einmal treffen muß. Betrachten Sie das folgende Beispiel:

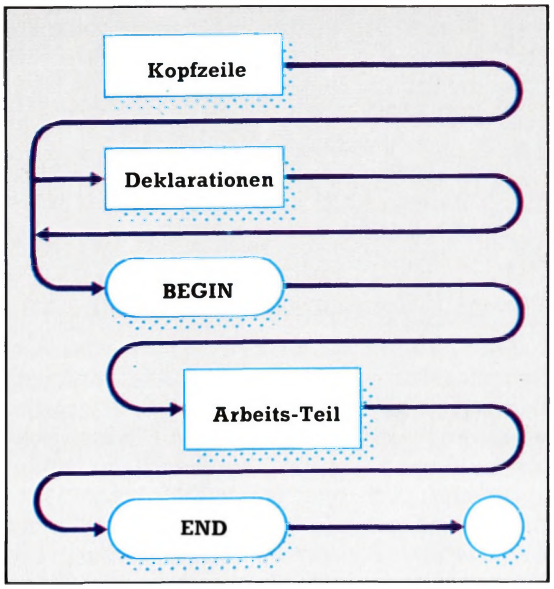
```
IF N = 1
   THEN
      write ('st.')
   ELSE
      IF N = 2
         THEN
            write ('nd.')
         ELSE
            IF N = 3
               THEN
                  write ('rd.')
               ELSE
                  write ('th.')
```

Wann immer die auszuführende Anweisung vom Wert einer einfachen Skala in einem begrenzten Bereich abhängt, kann man die CASE-Anweisung verwenden. Für das hier gezeigte Beispiel sieht das dann so aus:

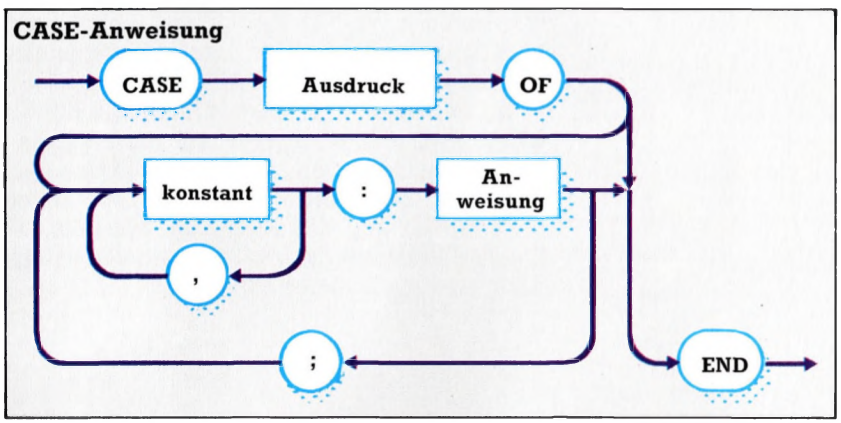
```
CASE N OF
   1              : write ('st.');
   2              : write ('nd.');
   3              : write ('rd.');
   4,5,6,7,8,9    : write ('th.')
END {CASE}
```

Beachten Sie, daß nur die Werte von N Gültigkeit haben, die auch innerhalb der CASE-Anweisung spezifiziert wurden – in diesem Beispiel die Werte 1 bis 9. Alle Werte, die N während der Programmausführung eventuell beinhalten kann, müssen angegeben werden. Für das hier gezeigte Beispiel wäre somit die Eingabe 0 für N nicht möglich.

Viele PASCAL-Compiler berücksichtigen auch die zusätzlichen reservierten Worte OTHERWISE oder OTHERS, die zur Aktivierung eines alternativen Programmteils verwendet werden können. Ausführliche Informationen über diese Wörter entnehmen Sie bitte dem Handbuch Ihrer PASCAL-Version.



Dies ist das vereinfachte Flußdiagramm eines PASCAL-Programms, in dem alle wesentlichen Elemente enhalten sind. Beachten Sie, daß die reservierten Wörter von PASCAL in Großbuchstaben geschrieben sind. Diese Form der Darstellung soll dabei helfen, sie von den nicht reservierten Wörtern zu unterscheiden. Die meisten PASCAL-Versionen akzeptieren jedoch auch reservierte Wörter in Kleinbuchstaben.



CASE ist die einzige Anweisung in PASCAL, die das Wort END zur Begrenzung einer Struktur verwendet, die nicht mit BEGIN gestartet wurde. Es ist daher allgemein üblich, das Ende jeder CASE-Anweisung in der hier gezeigten Form zu kennzeichnen.

Die Struktur arbeitet in der Form, daß sie den Skalar-Ausdruck zwischen den reservierten Wörtern CASE und OF berechnet. (Ein einfacher Variablenname ist ein Ausdruck, der keine Berechnungen erfordert.) Der ermittelte Wert wird mit den folgenden Konstanten verglichen. Trifft ein Vergleich zu, wird die dem Doppelpunkt folgende Anweisung ausgeführt. Die Programmkontrolle überspringt dabei keinen Teil. Dadurch bleibt der Zusammenhang der Konstruktion gewahrt. Wenn mehrere Operationen ausgeführt werden müssen, kann selbstverständlich auch noch eine zusammengesetzte Anweisung zwischen den Worten BEGIN und END verwendet werden.

Unter gewissen Umständen kann es vorkommen, daß einige Werte keinerlei Aktivität erfordern. In solchen Fällen kann die einfachste aller PASCAL-Anweisungen verwendet wer-

den. Diese Anweisung ist die sogenannte "Null"-Anweisung, die soviel bedeutet wie "mache nichts". Dazu ein Beispiel:

```
CASE N MOD 4 OF
  0: {mache nichts};
  1,3 : begin
    write (N MOD 4: 1, 'viertel');
    if N MOD 4 > 1 then
      write ('s')
    end;
  2 : write ('1/2')
END {CASE}
```

Beachten Sie, daß nach wie vor ein Semikolon benötigt wird, um auch diese nicht existierende Anweisung von folgenden Programmteilen zu trennen. Bei der Anweisung der letzten Zuordnung muß kein Semikolon gesetzt werden, da ein reserviertes Wort folgt (END) und keine weitere Zuordnung oder Anweisung. Die Verwendung des MOD-Operators gewährleistet, daß der Wert im Bereich von 0 bis 3 liegt. MOD gibt den Rest einer Integer-Division aus, ähnlich wie bei einigen BASIC-Versionen.

Im nächsten Teil dieses PASCAL-Kurses werden wir uns mit diesen und allen anderen Operatoren sowie mit eingebauten Funktionen befassen. Zum Abschluß zeigen wir Ihnen noch die Lösung des Problems, wie man die Zeichen-Strings für jeden Wert unseres selbstdefinierten Datentypes ausdruckt:

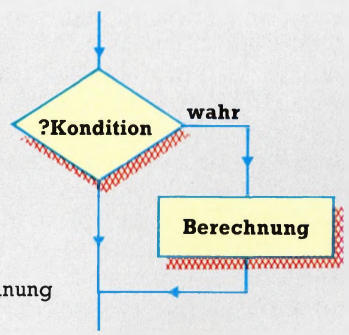
```
CASE farbwert OF
  rot      : write ('Rot');
  gruen    : write ('Gruen');
  gelb     : write ('Gelb');
  blau     : write ('Blau');
  magenta  : write ('Magenta');
  cyan     : write ('Cyan');
END {CASE}
```

Vergleichs-Konstruktionen

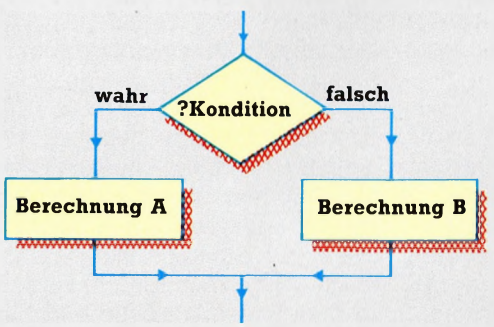
IF ...THEN führt den Programmteil „Berechnung" dann aus, wenn der Vergleich wahr ist. Ist der Vergleich falsch, so wird die Programmausführung einfach mit der nächsten Anweisung fortgesetzt.

Im Gegensatz zur IF ...THEN-Konstruktion, die eine „Option" anbietet, stellt IF ...THEN ...ELSE eine „Auswahl" zur Verfügung. Abhängig vom Ergebnis des Vergleichs wird eine von zwei Berechnungen ausgeführt.

Option
IF Kondition
  THEN Berechnung



Auswahl
IF Kondition
  THEN Berechnung A
  ELSE Berechnung B



Wieviele Tage bis Ultimo?

Das folgende Programm berechnet lediglich die Anzahl der Tage zwischen einem vom Anwender eingegebenen Datum und dem Ende eines Monats. Trotzdem können Sie hier einige Grundsätze der PASCAL-Programmierung erkennen.

Die CASE-Anweisung wird zur Umsetzung der eingegebenen numerischen Daten in das interne Datenformat verwendet. Anschließend werden die Daten wieder umgesetzt, damit sie als String auf dem Bildschirm ausgegeben werden können. Beachten Sie die Zeichen-Variable (symbol), die nur dazu dient, die beliebigen Zeichen zu lesen, die zur Trennung der Eingabe verwendet werden.

```
PROGRAM          Datum          (input,output);
CONST
     Punkt='.';
TYPE
    Kalender=(Jan, Feb, Mrz, Apr, Mai,
      Jun, Jul, Aug, Sep, Okt, Nov, Dez);
VAR
    Monatname      :Kalender;
    tag,
    monat,
    jahr,
    rest           :integer;
    symbol         :zeichen;
    Schaltjahr     :boolean;
BEGIN
  WriteLn ('Geben Sie das Datum wie folgt ein:');
  WriteLn ('TT/MM/JJ':40);
  WriteLn;
  write ('Datum?');
  read (tag, symbol, monat, symbol, jahr);
  Schaltjahr:=jahr MOD 4 = 0;
      {Jahrhunderte werden nicht beachtet}
IF (monat > 0) AND (monat < = 12)
  THEN
    CASE monat OF

    1  :MonatName:=Jan;     7  :MonatName:=Jul;
    2  :MonatName:=Feb;     8  :MonatName:=Aug;
    3  :MonatName:=Mrz;     9  :MonatName:=Sep;
    4  :MonatName:=Apr;     10 :MonatName:=Okt;
    5  :MonatName:=Mai;     11 :MonatName:=Nov;
    6  :MonatName:=Jun;     12 :MonatName:=Dez

    END              {CASE}
  ELSE
    BEGIN
      WriteLn ('Aeh?');
      WriteLn (' — das Programm kann nicht', 'weiter ausgefuehrt
          werden!')
    END; {MonatName ist nicht initialisiert}

  CASE MonatName OF
  Jan,Mar,
  Mai,Jul          :Rest:=31—tag;
  Apr,Jun
  Sep,Nov          :Rest:=30—tag;
  Feb              :IF Schaltjahr
                      THEN
                        Rest:=29—tag
                      ELSE
                        Rest:=28—tag

  END; {CASE}

  WriteLn;
  write('Es sind noch', Rest:1,
          'Tage im Monat');

  CASE Monat Name OF
    Jan            : write ('Januar');
    Feb            : write ('Februar');
    Mrz            : write ('Maerz');
    Apr            : write ('April');
    Mai            : write ('Mai');
    Jun            : write ('Juni');
    Jul            : write ('Juli');
    Aug            : write ('August');
    Sep            : write ('September');
    Okt            : write ('Oktober');
    Nov            : write ('November');
    Dez            : write ('Dezember');

  END; {CASE}

  WriteLn (Punkt)

  END.
```

Eins plus zwei gleich siebzig?

Mit seinen strukturierten Datenklassen und vielen Datentypen eignet sich PASCAL ideal zur Arbeit mit der Logik komplizierter mathematischer Abläufe.

Wieviel ist eins plus zwei? Ohne zusätzliche Information wird die Antwort automatisch „drei" lauten. Wir setzen dabei voraus, daß wir es mit natürlichen Zahlen zu tun haben. Wenn wir jedoch ein Fünfzigpfennigstück und zwei Zehnpfennigstücke in einen Automaten einwerfen, kann das Ergebnis durchaus siebzig Pfennig sein – oder, wenn man es anders betrachtet, eine Tasse Kaffee. Die genaue Angabe der Datenklasse hat folglich große Bedeutung.

In PASCAL können Sie nun eine ganze Reihe von Datentypen selbst definieren und haben damit die Möglichkeit, Ihre Programmalgorithmen exakt auf die eingesetzte Datenstruktur zuzuschneiden. Der PASCAL-Compiler meldet dabei sofort einen Fehler, wenn beispielsweise einer Booleschen Variablen über die Tastatur ein Wert zugeordnet werden soll. Diese Hilfe kann Stunden frustrierender Fehlersuche ersparen, da der Compiler kein Quellenprogramm ausführt, das noch Ablauffehler enthält. Einer der Vorteile von PASCAL ist, daß es zahlreiche unterschiedliche Datentypen verarbeitet und man diese auf verschiedene Arten beschreiben kann.

Die einfachen Variablentypen (die nur einen einzigen Wert enthalten können) arbeiten nach folgenden Regeln: Unterschiedliche Skalartypen sind untereinander nicht kompatibel. Es gibt jedoch zwei numerische Variablen, die viele gemeinsame Eigenschaften besitzen. So können sich reale Zahlen (Real) Ganzzahlen (Integer) annähern und Ganzzahlen in reale Zahlen umgewandelt werden. Reale Zahlen können jedoch nie völlig zu Ganzzahlen werden. Hier einige Beispiele:

```
Program Compatibility (input, output);
VAR
  intA,
  intB :integer;
  Xreal,
  Yreal :real;
BEGIN
  read (intA,Xreal); (*erst Integer, dann einen
    Real eingeben*)
  Yreal := intA; (*real := ist möglich*)
  intB := Xreal; (*FEHLER : Umwandlung
    nicht möglich*)
  (*etc.
```

Arithmetische Vorgänge sind logischerweise nur für numerische Variablen definiert. Die folgenden mathematischen Zeichen können mit numerischen Variablen vom Typ Integer und Real eingesetzt werden:

\+ Addition
– Subtraktion
* Multiplikation
/ Divison (Fließkommaformat)

Diese Zeichen werden „dyadische" (zum Zweiersystem gehörende) Operatoren genannt, da sie zwei Operanden (Real oder Integer) benötigen. Ist einer der Operanden ein Real, ergibt sich ein Real: 2 + 2.0 ist 4.0 (nicht 4). Das Ergebnis einer Division ist in jedem Fall ein Real, selbst wenn beide Operatoren Integer sind: 3/4 ist 0.6 und 8/4 ist 2.0.

Bei einer Division zweier Integers läßt sich mit einem Real als Ergebnis oft nichts anfangen. Verteilt man beispielsweise zwölf Stücke Schokolade an zehn Personen, so erhält jeder ein Stück, während zwei Stücke übrig bleiben. Mit den zwei Operatoren DIV und MOD (beides Schlüsselwörter in PASCAL) ist die Ganzzahlenteilung jedoch möglich. Das Ergebnis und der Rest werden dabei im Integerformat angezeigt: 15 DIV 5=3 und 15 MOD 5=0; 31 DIV 7=4 und 31 MOD 7=3. Dabei dürfen keine negativen Werte eingesetzt werden, da die Integerdivision für negative Nenner nicht definiert ist. Bei beiden Divisionsarten – Real und Integer – ist die Teilung durch Null ein Fehler, da noch niemand herausgefunden hat, wie die Unendlichkeit bewertet werden kann.

Multiplikation und Division werden vor Addition und Subtraktion ausgeführt. Mit Klammern läßt sich diese Rangfolge jedoch ändern. (8+4) DIV 2=6 aber 8+4 DIV 2=10.

Obwohl Reals nicht unmittelbar in Integers umgewandelt werden können, führen „Trunc" und „Round" diese Funktion indirekt aus. Trunc schneidet bei einem Real alle Stellen hinter dem Komma ab: trunc (3.999) = 3 und trunc (–123,456) = –123. Die Funktion round rundet eine Zahl gegen Null ab, wenn die Nachkommastellen kleiner sind als 0.5. Liegen sie darüber, wird in die andere Richtung gerundet: round (1234.5) = 1235 und round

(−0.49237) = 0. Ein Fehler tritt auf, wenn das Ergebnis der beiden Funktionen außerhalb des für den Typ Integer verfügbaren Zahlenbereiches liegt (−MaxInt bis MaxInt). Fangen Sie diese Möglichkeit sicherheitshalber vorher mit einer IF-Abfrage ab. Die in Klammern stehende Zahl muß ein Real sein, da Integers keine Nachkommastellen besitzen und eine Rundung nicht möglich ist.

Die Funktion odd bietet in PASCAL eine weitere Möglichkeit, Zahlenwerte zu überprüfen. Damit läßt sich feststellen, ob ein Integer gerade oder ungerade ist. Odd erzeugt ein Ergebnis vom Typ der Booleschen Variablen: Ist die in Klammern stehende Zahl gerade, ergibt odd den Wert False; bei ungerade den Wert True.

```
IF odd(N)
   THEN
      WriteLn ("ungerade")
   ELSE
      WriteLn ("gerade")
```

oder:

```
IF odd(N) THEN
   IF N MOD 2=0 THEN
      WriteLn ("Der Compiler spinnt!")
```

In der kleinen Tabelle auf der nächsten Seite sind alle arithmetischen Funktionen von PASCAL aufgeführt. Die jeweils in Klammern stehende Variable kann jeden Wert vom Typ Real und Integer annehmen, wobei sich das Ergebnisformat von abs und sqr dem Klammerwert anpaßt: abs(−19.372) ist 19.372 und abs(255) ist 255. Alle anderen Funktionen erzeugen Reals: sqrt(16) ist 4.0, nicht 4. Der Grund dafür sind die Berechnungsalgorithmen. Sie bauen auf der Summierung von Reihen auf und arbeiten mit Kommazahlen.

Sie werden bemerkt haben, daß wir bei Ergebnissen vom Typ Real das Wort „ist" verwenden und nicht das Gleichheitszeichen; „ist" bedeutet hier „ist dasselbe wie" statt „ist gleich". Reals sollten nie auf exakte Gleichheit hin überprüft werden, da schon der kleinste Berechnungsfehler bewirken kann, daß zwei formal „gleiche" Reals sich um 1.0E-27 unterscheiden. Für uns ist dieser Unterschied zwar sehr klein, für den Computer jedoch nicht. Statt IF X = Y THEN ... können Sie die abs-Funktion folgendermaßen verwenden:

```
IF abs(X—Y) < Kleinste Differenz
THEN (* etc.*)
```

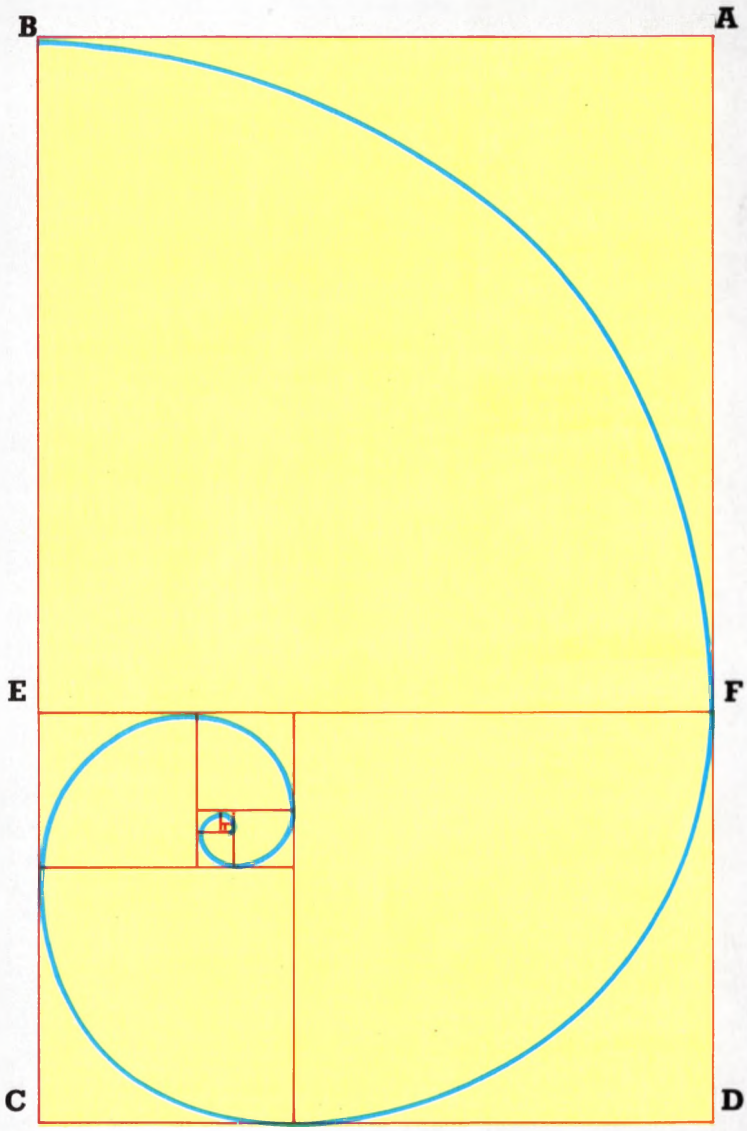
Hierbei wird getestet, ob die Differenz zwischen X und Y unter einem bestimmten Minimalwert liegt, der am Anfang des Programms über CONST definiert ist. Logarithmen werden zur Basis (e) angegeben, nicht zur Basis 10, während exp den Wert e (2,71828...) um die in Klammern stehende Zahl potenziert. Wirth entschied sich ganz bewußt dafür, in PASCAL keine Potenzfunktion aufzunehmen. In BASIC-Programmen findet man oft Zeilen wie

```
500 LET D = B^2+4*A*C
```

Der goldene Schnitt

Eine Gerade läßt sich nur an einem einzigen Punkt derart teilen, daß das Verhältnis der kürzeren Strecke zur längeren Strecke dem Verhältnis der längeren Strecke zur ganzen Linie gleicht. Dieses Verhältnis wird „sectio aurea" oder „goldener Schnitt" genannt und ist – wie Pi – eine irrationale Zahl mit unendlich vielen Nachkommastellen.

Der goldene Schnitt wurde besonders von griechischen Künstlern und Architekten eingesetzt. Einige der interessantesten Beispiele finden sich jedoch in der Natur. So baut die „Seespirale" ihre Schale nach dem goldenen Schnitt. Die Radiusvektoren – nach jeweils 90 Grad aufgezeichnet – teilen einander in diesem Verhältnis.



Die Seiten des Rechtecks ABCD werden nach dem goldenen Schnitt geteilt. Ein Quadrat ABEF kann darin derart untergebracht werden, daß sich ein Rechteck (EFDC) mit dem gleichen Seitenverhältnis wie ABCD bildet. Dieser Vorgang läßt sich bis ins Unendliche fortsetzen.

die das Quadrat einer Zahl nach der langsamsten und ungenauesten Methode bilden. B * B eignet sich dafür weitaus besser. In PASCAL ergibt sqr(N) das Quadrat von N im Integerformat (vorausgesetzt, N ist ein Integer); sqr(X/3) erzeugt ein Real. Und noch eine BASIC-Gewohnheit muß aufgegeben werden: sqrt(x) und nicht sqr(x) ergibt die Quadratwurzel einer Zahl.

Alle Skalartypen haben einen festen und geordneten Wertebereich. Von ihnen können Untertypen abgeleitet werden, die einen Teilbereich des ursprünglichen Skalartyps umfassen. Bei der Definition dieser Untertypen werden die Unter- und Obergrenzen in Klammern angegeben:

```
TYPE byte = 0..255; (*Unterbereich von
   Integer*)
   alpha = 'A'..'Z'; (*Unterbereich von char*)
   farbe = (kreuz, pik, herz, karo); (*ein
   neuer Typ*)
   hauptfarbe =(herz . . karo);(*Unterbereich
   von farbe*)
```

Außer den erwähnten Vorteilen kann ein guter PASCAL-Compiler Variablen vom Typ byte beispielsweise in acht Bytes (bei „gepackter" Darstellung) statt in 32 Bytes unterbringen. Eine Matrix dieser Art belegt nur 25 Prozent der Speicherkapazität, die normale Integerelemente einnehmen würden. Beachten Sie, daß der Typ hauptfarbe zwei mögliche Werte enthält, die nicht vom Booleschen Typ sind, wie:

```
TYPE
   boolean = (false, true);
```

Alle Untertypen übernehmen Datenstruktur und Abläufe des Haupttyps.

Die Definition von alpha kann außer dem Alphabet natürlich auch andere Zeichen umfassen. So enthält der ASCII-Zeichensatz zwischen „Z" und „a" sechs weitere Symbole, darunter die eckigen Klammern („[" und „]"), die zu den 23 Schlüsselwörtern von PASCAL gehören. Sie dienen – wie in vielen anderen Sprachen – als Grenzmarkierung für die Komponenten mehrerer strukturierter Datentypen. Auch BASIC hatte diese Symbole ursprünglich für Matrixindizes eingesetzt. Da jedoch einige ältere Computer nur über den „halben" ASCII-Zeichensatz ohne Kleinbuchstaben, eckige und geschweifte Klammern und einige andere Symbole verfügten, wurde die Syntax auf runde Klammern umgestellt. Der ursprüngliche Apple II war auch an diese Grenzen gebunden. PASCAL bietet folgende Alternativen: (. und .) lassen sich für [und] einsetzen und (* und *) statt { und }. Diese alternative Kennzeichnung von Kommentaren wird oft verwandt, wobei nur bestimmte Compiler die eckigen Klammern verarbeiten können. Die Begrenzungen lassen sich auch mischen: (* Dies ist ein korrekter Kommentar }.

Funktion	Ergebniswert
abs(K)	Absolutwert von K
sqr(K)	Quadrat von K
sqrt(K)	Quadratwurzel aus K
sin(A)	Sinus von A (A im Bogenmaß)
cos(A)	Cosinus von A (A im Bogenmaß)
arctan(T)	Arcus Tangens von T (Ergebnis im Bogenmaß)
ln(K)	Logarithmus von K zur Basis e
exp(L)	Wert von e hoch L

Das goldene Band

Das hier aufgeführte Programm erzeugt Fibonacci-Einheiten und zeigt sie mit den entsprechenden Verhältniszahlen an. Interessanterweise nähert sich dabei das Verhältnis jedes aufeinanderfolgenden Einheitenpaars mehr und mehr den Werten des goldenen Schnittes.

Das Programm enthält aber auch einige Grundlagen, die wir in früheren Folgen dieser Serie untersucht hatten, darunter weitere Beispiele für die REPEAT-Struktur von PASCAL. Versuchen Sie einmal, die Endbedingung der Schleife so zu verändern, daß das Programm endet, wenn die nächste zu berechnende Fibonacci-Einheit größer als MaxInt ist.

```
PROGRAM     Golden (output);

CONST    Epsilon          = 1.0E-08;

TYPE     Fibonacci        = 1..MaxInt;

VAR

first,
second,
next        : Fibonacci;
ratio,
Gold        : real;
count       : integer;

BEGIN
  WriteLn ('Der goldene Schnitt' : 30);
  WriteLn;
  WriteLn ('Fibonacci Serie:');
  first := 1; (* per Definition*)
  second := 1;
  ratio := first/second;
  count := 0;

REPEAT
  IF count MOD 10 = 0 THEN
    BEGIN (*alle 10 Zeilen — Druck der Ueberschrift*)
      WriteLn;
      WriteLn ('First' : 10,
        'Second' : 10, 'Ratio' : 14);
      WriteLn
    END;

  WriteLn (first : 10, second : 10,
    ratio : 16 : 8);
  count := count + 1;

  Gold := ratio; (* alter GS*)
  next := first + second;
  first := second; (* nachstes Element der *)
  second := next; (*Serie aufrufen*)
  ratio := first/second

UNTIL abs (ratio - Gold) < Epsilon;

WriteLn;

WriteLn ('die goldene Mitte ist :',
    100 * ratio : 10 : 5 '%')

END.
```

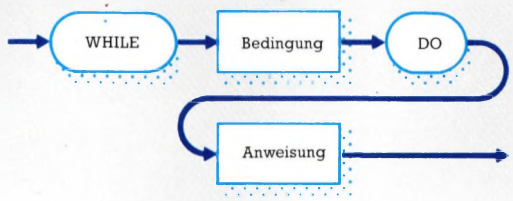
Mach's noch einmal, RAM

In unserer Einführung in PASCAL wurden bisher die strukturierten Anweisungen gezeigt, mit denen sich Programme exakt auf einzelne Probleme zuschneiden lassen. Heute untersuchen wir drei verschiedene Strukturen für die Steuerung von Programmschleifen.

Zwei der drei Möglichkeiten zur Strukturierung von PASCAL-Anweisungen haben wir bereits dargestellt. Sequentielle Strukturen sind in die Wörter BEGIN und END eingeschlossen, während bedingte Anweisungen mit IF und CASE aufgebaut werden. Für den dritten Strukturtyp – die Wiederholung – stellt PASCAL drei Konstruktionsmöglichkeiten zur Verfügung. Die FOR-Anweisung kann eine feste Anzahl von Schleifendurchgängen aufrufen, während WHILE und REPEAT durch Bedingungen gesteuert werden.

In der WHILE-Schleife wird zunächst der Boolesche Ausdruck zwischen den beiden reservierten Wörtern WHILE und DO bewertet.

Die WHILE-Anweisung:



Ist das Ergebnis wahr („true"), dann wird die folgende Anweisung (die jede nur mögliche PASCAL-Anweisung mit beliebig gestaffelter Tiefe sein kann) so lange wiederholt, wie die Bedingung zutrifft. Nach jeder Ausführung der Anweisung wird die Boolesche Bedingung des WHILE-Befehls neu bewertet. Das bedeutet aber auch, daß mindestens einer der in der Anweisung enthaltenen Werte durch die Abläufe innerhalb der Schleife verändert werden muß. Hier ein Beispiel:

```
WHILE MaxInt > 1 DO
  WriteLn ('Schleife wird ausgefuehrt')
```

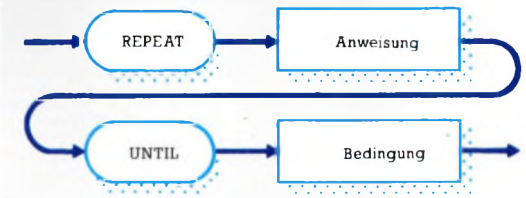
Da diese Schleife sich nur schwer beenden läßt, hier ein realistischeres Beispiel:

```
read ( Abhebung );
WHILE Abhebung > KontoGuthaben DO
  BEGIN
    WriteLn ('Nicht genug Geld vorhanden —
      Neuer Versuch:');
    write ('Abhebung DM ?'),
    read ( Abhebung )
END
```

Dieser Programmteil verdeutlicht das wichtigste Merkmal der WHILE-Struktur: Wenn die Abhebung kleiner ist als das augenblickliche Kontoguthaben, wird die WHILE-Schleife nicht ausgeführt. Die Ausführung der Schleife wird beendet, wenn die Bedingung unwahr („false") ist. Die Steuerung wird dann wieder von dem sequentiellen Ablauf übernommen, und die erste Anweisung wird nach der WHILE-Struktur ausgeführt.

Beachten Sie, daß im Gegensatz zu WHILE die Endbedingung für REPEAT..UNTIL am Ende der Struktur liegt. Die REPEAT-Schleife wird beendet, wenn der Boolesche Ausdruck „true" ist, während die WHILE-Bedingung nur dann aufhört, wenn die Bedingung „false" ist. Sind innerhalb der REPEAT-Struktur mehrere Anweisungen enthalten, können sie in BEGIN..END eingeschlossen werden. Dies ist jedoch nicht nötig, da die reservierten

Die REPEAT-Anweisung:



Wörter REPEAT und UNTIL als Begrenzung ausreichen. BEGIN und END haben die gleiche Funktion wie ein zusätzliches Semikolon vor einem reservierten Wort, wobei jedoch keine leere Anweisung entsteht.

Der folgende Programmteil zählt, wie oft der Buchstabe e in einem per Tastatur eingegebenen Text vorkommt. Die Eingabe muß mit einem Punkt abgeschlossen werden, damit die REPEAT-Schleife beendet und das Ergebnis dargestellt werden kann.

```
Zahl:= 0 ;
REPEAT
  read (Zeichen) ;
  IF Zeichen = 'e' THEN
    Zahl :=Zahl + 1
UNTIL Zeichen = '.';
WriteLn ('In dem Satz sind ', Zahl : 1,
  '''e''s enthalten.')
```

Antworten zu den Übungen

In der vorigen Folge stellten wir die Aufgabe, bei der Generierung der Fibonacci-Einheiten und der Verhältniszahlen der aufeinanderfolgenden Paare die Schleife so zu verändern, daß die Endbedingung eintritt, wenn die nächste Fibonacci-Einheit, die berechnet werden soll, MaxInt überschreitet.

Eine Ganzzahl, die bei der Darstellung im Zweierkomplement MaxInt überschreitet, zerstört das eigene Vorzeichenbit. Da die Zahl dann negativ erscheint, würde das Programm „abstürzen". Es kann keine negativen Werte verarbeiten und würde selbst mit ganzzahligen Werten die Schleife niemals beenden können:

```
UNTIL first+second >
  MaxInt
```

kann nie eintreten, da es für den Computer keine Zahl gibt, die größer als MaxInt ist. Mit einem kleinen Trick läßt sich das Problem jedoch lösen:

```
UNTIL second >
  MaxInt-first
```

Jetzt werden natürlich auch die Real-Arithmetik und die Konstante Epsilon nicht mehr benötigt.

Für diesen Ablauf lassen sich beide Strukturen einsetzen. Die WHILE-Version sieht folgendermaßen aus:

```
Zahl:= 0 ;
read (Zeichen) ;
WHILE Zeichen <> '.' DO
  BEGIN
    IF Zeichen = 'e' THEN
      Zahl := succ (Zahl);
    read (Zeichen) ;
  END;
write ('In dem Satz sind ', Zahl : 1, );
WriteLn ('''e''s enthalten.')
```

WHILE-Befehlseingabe

Der Unterschied zwischen beiden Strukturen ist leicht zu erkennen. In der WHILE-Struktur muß ein Befehl unter Umständen zweimal eingegeben werden (die read-Anweisung), wenn die Endbedingung eintreten soll, – einmal unmittelbar vor der WHILE-Struktur und einmal in der letzten Anweisung der Schleife.

Obwohl die WHILE-Struktur sehr viele Anforderungen erfüllt, ist es manchmal notwendig, daß ein Ablauf nach einer festen Anzahl von Wiederholungen – beispielsweise nach Durchlauf einer begrenzten Werteskala – beendet werden kann. PASCAL bietet dafür die FOR-Struktur, die jedem BASIC-Programmierer vertraut ist. Zwischen den FOR-Schleifen beider Sprachen bestehen jedoch wesentliche Unterschiede. Zunächst kann jeder Skalar die Steuerung der Schleife übernehmen, nicht nur eine Ganzzahl. Wichtiger ist jedoch die Präzision der Anweisung. Wie die WHILE-Schleife kann auch FOR . .DO nicht ausgeführt werden, wenn beispielsweise der Anfangswert größer ist als der Endwert. Außerdem sind die Steuervariablen in PASCAL sehr starken Einschränkungen unterworfen. Insbesondere läßt sich ihr Wert nicht innerhalb der Schleife verändern – schon der Versuch wird von PASCAL als Fehler gemeldet. Der Vorteil dieser Sicherheitsmaßnahmen wird noch deutlich werden. Zunächst jedoch ein illegales Beispiel:

```
FOR N := 1 TO 10 DO
    IF N = 10 THEN
      N := 1
```

Diese Struktur ist fehlerhaft, da die IF-Anweisung nach zehnfacher Ausführung der Schleife die Variable N wieder auf 1 setzt und so eine unendliche Schleife erzeugt.

Die Syntax der FOR . .DO-Anweisung verlangt zwischen den beiden Begrenzungen (FOR und TO) die Zuweisung des Anfangswertes der Steuervariablen. Der Endwert wird von den reservierten Wörtern TO und DO eingeschlossen. Hier einige Beispiele:

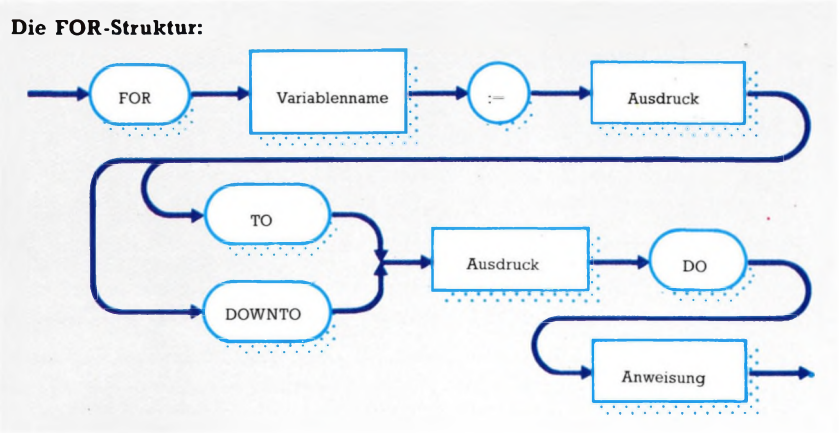
```
FOR Zeichen := 'A' TO 'Z' DO (* etc. *)
FOR Monat := 'Jan' TO 'Dez' DO (* etc. *)
FOR N := N TO succ(MaxInt DIV 1000) DO
  (* etc. *)
```

Das letzte Beispiel setzt voraus, daß N zuvor schon ein geeigneter Wert zugeordnet wurde. Wenn dieser Wert den höchstmöglichen Wert überschreitet, wird die Schleife überhaupt nicht ausgeführt. Für eine absteigende Wertskala wird anstelle von TO das reservierte Wort DOWNTO verwandt. So könnte der Start einer Rakete folgendermaßen programmiert werden:

```
FOR CountDown := 10 DOWNTO 0 DO
  WriteLn (Countdown : 32 — 3 *
    CountDown);
WriteLn ('Wir haben LIFT OFF !')
```

Die Ausführung der FOR-Schleife wird von PASCAL sehr streng gehandhabt. Es gibt keine Entsprechung für den BASIC-Befehl STEP, der die Inkrementierung und Dekrementierung modifizieren kann.

Die FOR-Struktur:



Bierpreise

```
PROGRAM     BierPreise     (output);
Type
  Bier = ( Hell, Dunkel, Bock, Alt );
VAR
  Preis,
  Halbe : real ;
  Typ   : Bier ;
BEGIN
  WriteLn ( 'Halbe' :7,
    'Hell' :8, 'Dunkel' :9
      'Bock' :7, 'Alt' :7 ) ;
  WriteLn;
  Halbe :=0.5 ; (* Anfang bei einem Viertel *)
REPEAT
  IF Halbe — trunc ( Halbe ) > 0.4
    THEN (* als # #.# ausgeben *)
      write ( Halbe : 6 : 1, ' ' )
    ELSE (* als Ganzzahl ausgeben *)
      write ( round ( Halbe ) : 4, ' ' : 3 ) ;
  FOR Bier := Hell TO Alt DO
    BEGIN
      CASE Bier OF
        Hell   : Preis := 1.55;
        Dunkel : Preis := 1.65;
        Bock   : Preis := 1.75;
        Alt    : Preis := 1.85
      END; (* CASE *)
        (* runden und in Mark umwandeln *)
      write ( round ( Preis * Halbe )
        / 100 : 8 : 2 )
  END;
      (* neue Zeile anfangen *)
    WriteLn;
    Halbe := Halbe + 0.5
  UNTIL Halbe > 10
END.
```

Das Programm Bierpreise druckt in Viertellitersprüngen eine Preisliste für vier verschiedene Biersorten. Jede Biersorte ist in der Typendeklaration TYPE als Typenname aufgeführt. Die Auswahl der entsprechenden Preise geschieht mit der CASE-Anweisung. „Ganzzahlige" Halbe werden als Integer angezeigt und mit zwei Leerzeichen versehen. Durch die Formatierung der realen Zahlen (reals) verschwinden unerwünschte Dezimalstellen.

Sets und Mengen

In PASCAL gibt es außer den Datenstrukturen Array und File noch Sets und Records. In dieser Folge untersuchen wir, wie Sets funktionieren, und geben einen Überblick, welche der PASCAL-Operatoren Vorrang vor anderen haben.

Wir haben uns zwar oft mit dem Zeichensatz eines Computers beschäftigt, aber nie genau definiert, was der Unterschied zwischen einem Satz (Set) und einem Array ist. Ein Set (oder auch Menge) ist eine Ansammlung von Objekten, die als Einheit angesprochen wird und nicht über die einzelnen Elemente. Da Sets keine strukturierte Ordnung haben müssen, ist oft nur die Frage wichtig: Ist ein bestimmtes Objekt Element des Sets oder nicht?

Aus Gründen der Einsatzfähigkeit sind Sets in PASCAL bestimmten Beschränkungen unterworfen. So können Sets nur einfache Skalare als Elemente haben und keine Arrays oder andere Datenstrukturen. Außerdem wird dem „Basistyp" eines Sets beim Einsatz in einem Programm eine Obergrenze gesetzt. Die Definition eines Sets ist einfach und direkt:

```
TYPE
  Zahlen   = SET OF 0..127;
  Alphabet = SET OF 'A'..'Z';
  Farben   = SET OF (Rot, Gruen, Blau);
```

Der Wertebereich, der für den ordinalen Basistyp eines Sets möglich ist, wird wie ein normaler Unterbereichstyp angegeben.

Set-Variablen werden wie alle anderen Variablen mit VAR deklariert. Ihre Elemente können – in eckige Klammern eingeschlossen – direkt angegeben werden:

```
VAR
  Codes  : Zahlen;
  Palette : Farben;
BEGIN
  Codes  := [0..2,4,8,16,32,64];
  Palette := [Rot..Blau]; (* etc. *)
```

Dieser Programmteil ordnet dem Set „Codes" die Zahlen 0 bis einschließlich 2 und 4,8,16,32 und 64 zu. Da dieser Set keine bestimmte Ordnung enthält, könnten wir ihn auch als [64,32,16,8,4,0..2] schreiben. Die Angabe 2..0 wäre als Definition eines Unterbereichstyps zwar illegal, würde ansonsten aber eine Leermenge bezeichnen. Es ist auch möglich, Sets über die Anweisung JederSet :=[] mit der Leermenge dieses Typs zu initialisieren. Dies ist in PASCAL die einzige Ausnahme der Regel, daß sich der Typ jedes Elements per Test feststellen läßt. Ohne mindestens ein Set-Element kann der Compiler den Typ allerdings nicht bestimmen.

Einer der praktischsten Operatoren für Set-Strukturen ist das reservierte Wort „IN". Mit IN läßt sich testen, ob ein Objekt (oder Objekte) Element eines Sets ist. IN ist ein Vergleichsoperator, der zwei Operanden benötigt. Dabei muß auf der linken Seite ein Ausdruck stehen, der zu den Set-Elementen gehört, und auf der rechten Seite eine Set-Variable oder eines oder mehrere Elemente des gleichen Set-Typs. Das Ergebnis ist der Boolesche Wert „true", wenn das Objekt ein Element der Vergleichsmenge ist, und „false", wenn dies nicht der Fall ist.

„N IN Codes" und „Gruen IN Palette" ergeben daher legale Boolesche Werte. Ein Test, ob ein Zeichen eine Zahl ist, wie:

```
IF (c>='0') AND (c< ='9') THEN . . .
```

läßt sich in PASCAL viel einfacher durchführen. Es muß nur festgestellt werden, ob c Element eines bestimmten Zeichensatzes ist:

```
IF c IN ('0'..'9') THEN . . .
```

So könnte der Anfang eines Kartenspielprogramms folgendermaßen aussehen:

```
TYPE
  Rang = ( Zwei, Drei, Vier, Fuenf, Sechs,
           Sieben, Acht, Neun, Zehn, Bube,
           Dame, Koenig, As);
  Kartenspiel : SET OF Rang;
VAR
  Karte  : Rang;
  Bilder : Kartenspiel;
BEGIN
  Bilder := (Bube . . As);
  IF Karte IN Bilder THEN (* etc. *)
```

Vergleichen Sie diese Anordnung mit:

```
IF (Karte = Bube) OR (Karte = Dame) OR
  (Karte = Koenig) OR . .
```

Ebenfalls definiert sind Operationen mit vollständigen Sets: Vereinigungsmenge, Mengendifferenz und Schnittmenge. Wenn B beispielsweise der Set aller BASIC-Programmierer ist und P der aller PASCAL-Programmierer, dann ist die Schnittmenge beider Sets der Set aller Programmierer, die sowohl BASIC als auch PASCAL beherrschen. Die Vereinigungsmenge von P und B ist der Set aller Menschen, die entweder in PASCAL oder in BASIC programmieren – also die Kombination beider Sets.

Eine Mengendifferenz ergibt sich aus der Subtraktion aller Elemente eines Sets von einem anderen. P–B stellt daher alle Personen dar, die in PASCAL programmieren, aber nicht

in BASIC. Die Vereinigungsmenge wird als P+B geschrieben und die Mengendifferenz als P*B. Die Symbole der Set-Operatoren gleichen zwar den bekannten arithmetischen Operatoren, die unterschiedlichen Vorgänge sollten jedoch nicht durcheinandergebracht werden. Da diese Symbole den Vorgang des Bit-Testens darstellen, werden sie auch für Sets eingesetzt.

Stellen Sie sich ein Set mit acht Elementen vor. Dabei wird die Anwesenheit eines Elementes durch das Setzen eines bestimmten Bits (Eins) in einem Byte angezeigt, seine Abwesenheit dagegen durch Null. Mit einer Maske (oder einem Bit-Test) kann nun leicht festgestellt werden, ob ein Byte Element des Sets ist. OR ergibt die Vereinigungsmenge und AND die Durchschnittsmenge. Da diese Abläufe auf Maschinenebene vorhanden sind, kann der PASCAL-Compiler Sets und ihre Operationen direkt in den Maschinencode umsetzen. Dafür wird nur wenig Speicher benötigt. Daher gehören Set-Operationen zu den effektivsten Abläufen in PASCAL.

Behandlung der Operator

Die Prioritätsregeln der PASCAL-Operatoren lassen sich auf vier Ebenen zusammenfassen, wobei die Operatoren, auf die wir nicht ausführlich eingegangen sind, dem in anderen Programmiersprachen üblichen Gebrauch entsprechen. Die „monadischen" (oder einwertigen) Vorzeichenoperatoren + und – und der logische Operator NOT haben erste Priorität. Auf der zweiten Prioritätsebene befinden sich alle Multiplikationsoperatoren (darunter auch die Divisionssymbole). Die dritte Ebene enthält Addition und Subtraktion, und auf der letzten Ebene werden alle Vergleichsoperationen – darunter auch IN – durchgeführt.

Beachten Sie, daß die beiden logischen Operatoren AND und OR wie Multiplikations- bzw. Additionsoperatoren behandelt werden und damit die Boolesche Algebra korrekt umsetzen. Viele Vergleichstests müssen die Prioritäten daher mit Klammern umgehen. So erzeugt etwa IF N>0 AND N< 10 THEN . . . eine Fehlermeldung, da der Ausdruck 0 AND N zuerst bewertet wird und als Versuch gilt, zwei ganzzahlige Operanden mit einem logischen Operator zu verknüpfen.

Alle Operatoren der gleichen Priorität werden von links nach rechts abgearbeitet. Das Zuordnungssymbol (:=) hat eine geringere Priorität als die anderen Operatoren, da der Ausdruck auf der rechten Seite einer Zuweisung vor der Zuordnung erst vollständig bewertet werden muß. Noch ein Wort der Vorsicht: Gehen Sie niemals davon aus, daß ein Ausdruck nicht bewertet wird. Für den Fall N=0 läßt IF (N>0) AND K N<19) THEN . . . das Programm absturzen, da eine Teilung durch Null nicht möglich ist.

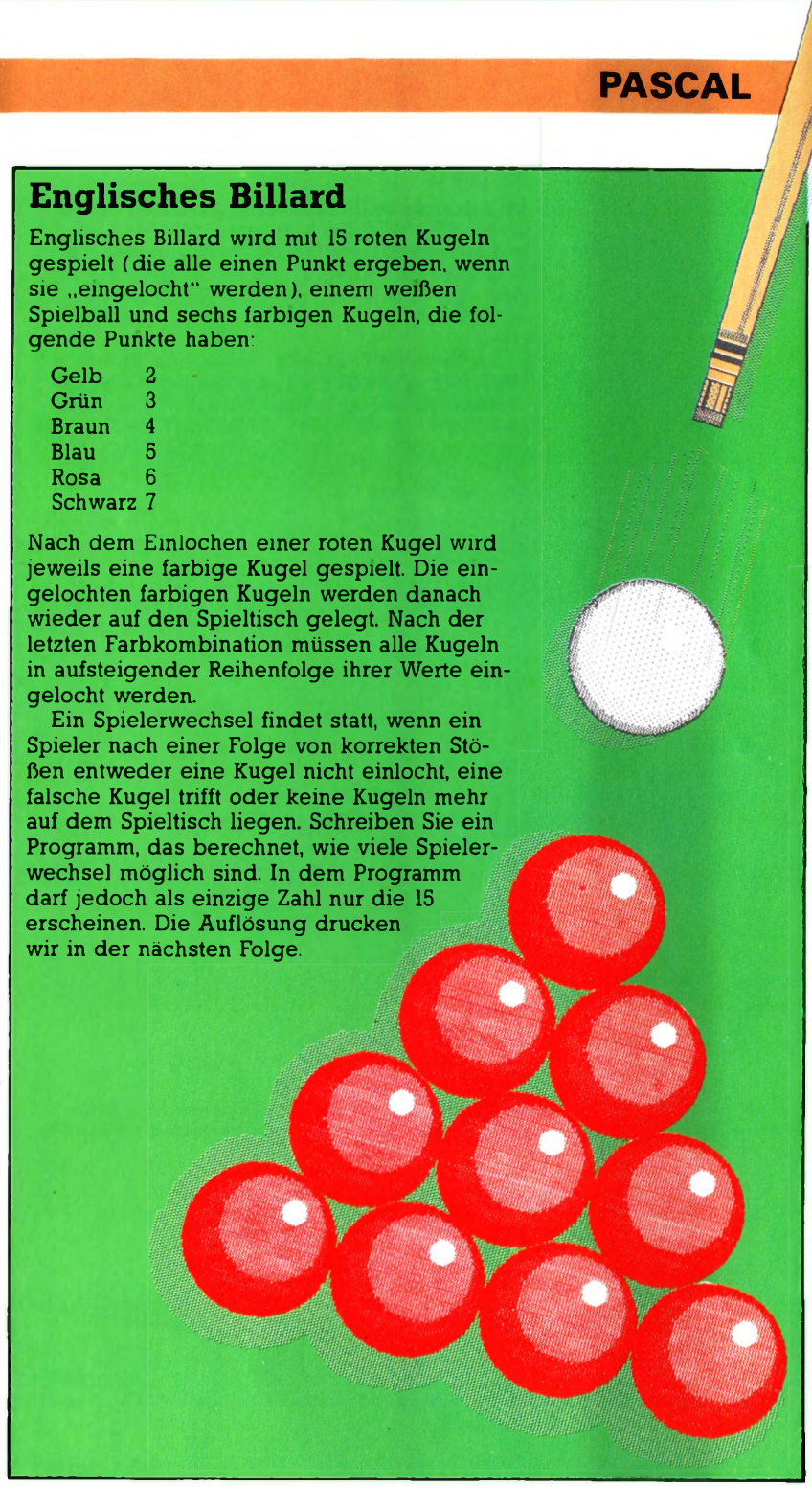
Englisches Billard

Englisches Billard wird mit 15 roten Kugeln gespielt (die alle einen Punkt ergeben, wenn sie „eingelocht" werden), einem weißen Spielball und sechs farbigen Kugeln, die folgende Punkte haben:

Gelb	2
Grün	3
Braun	4
Blau	5
Rosa	6
Schwarz	7

Nach dem Einlochen einer roten Kugel wird jeweils eine farbige Kugel gespielt. Die eingelochten farbigen Kugeln werden danach wieder auf den Spieltisch gelegt. Nach der letzten Farbkombination müssen alle Kugeln in aufsteigender Reihenfolge ihrer Werte eingelocht werden.

Ein Spielerwechsel findet statt, wenn ein Spieler nach einer Folge von korrekten Stößen entweder eine Kugel nicht einlocht, eine falsche Kugel trifft oder keine Kugeln mehr auf dem Spieltisch liegen. Schreiben Sie ein Programm, das berechnet, wie viele Spielerwechsel möglich sind. In dem Programm darf jedoch als einzige Zahl nur die 15 erscheinen. Die Auflösung drucken wir in der nächsten Folge.



Prioritätsregeln

Die Tabelle zeigt die Prioritäten der PASCAL-Operatoren. In runde Klammern eingeschlossene Ausdrücke werden separat bewertet und können so die normalen Prioritäten umgehen.

Priorität	Operator	Typ
Höchste	NOT + –	monadisch
	AND * / DIV MOD	Multiplikation
	OR + –	Addition
Niedrigste	< <= <> >=> IN	Vergleich

Bingo!

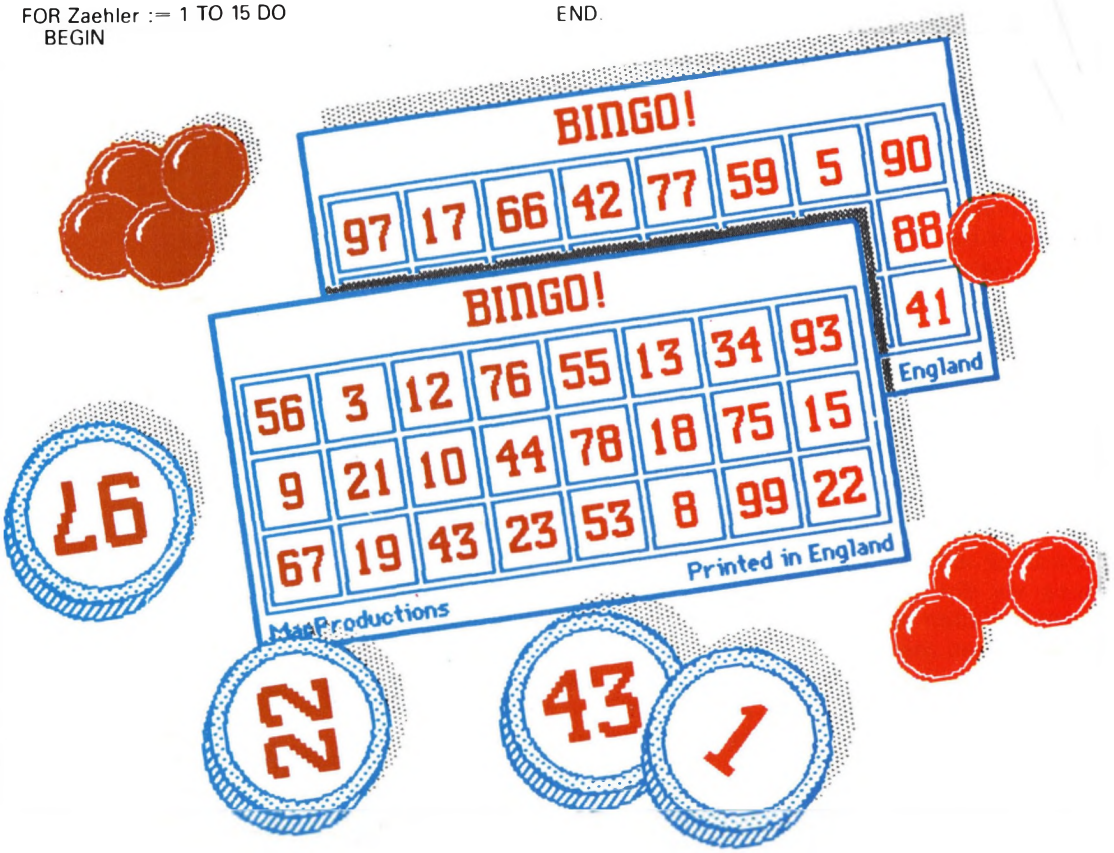
Eine Bingokarte läßt sich leicht mit einem Set darstellen. Obwohl die Grundelemente (Ganzzahlen von Eins bis 90) eine Struktur haben, ist hier das einzig wichtige Kriterium, ob die Nummer auf der Karte enthalten ist oder nicht. In PASCAL hieße das: „Nummer IN Karte" ist entweder „true" oder „false".

Das folgende Programm simuliert ein Bingospiel. Es liest zuerst die Kartennummern über die Tastatur ein und testet dann bei Aufruf der einzelnen Nummern, ob in diesen Nummern der Unterset Karte enthalten ist. Der Aufruf wird zur Leermenge, wenn sich jedes Element von Karte ebenfalls im Set „Aufruf" befindet.

Wenn zu einem Set ein Element hinzugefügt wird, das darin bereits enthalten ist, oder ein Element herausgenommen wird, das es darin nicht gibt, bleibt dieser Set unverändert. Beachten Sie, daß sich Elemente nicht direkt in einen Set einfügen lassen. Durch den Aufbau eines zusätzlichen Sets, der nur ein Element enthält, kann jedoch die Vereinigungsmenge dieser beiden Sets hergestellt werden.

Das Programm nimmt doppelte Nummerneingaben an und erzeugt einen Ablauffehler, wenn Zahlen außerhalb von Eins bis 90 eingegeben werden. Fügen Sie als Übung Schleifen ein, die dies verhindern und die die Eingabe bereits aufgerufener Nummern ablehnen.

```
PROGRAM                          Bingo (input, output);

CONST

                        Spalten = 40; (* Monitoranzeige *)
                        Haelfte = 25; (* Monitoranzeige *)

TYPE

                                 Bingo = SET OF 1 . . 90;

VAR
          Zaehler  : 1 . . 15;
          moeglich,
          Aufruf,
          leer,
          Karte    : Bingo;
          Nummer   : integer;
          Haus     : boolean;

BEGIN

  Leer := [ ] ;
  moeglich := [1 . . 90] ;
  WriteLn ( '*** BINGO ***' : Haelfte );
  WriteLn;
  WriteLn ( 'Geben Sie 15 Nummern der Karte ein,' );
  WriteLn ( 'Nach jeder Nummer bitte ein RETURN) :' );
  Karte := Leer;

  FOR Zaehler := 1 TO 15 DO
    BEGIN
      write ( Zaehler : 10, ': ? ' );
      ReadLn ( Nummer );
      Karte := Karte + [Nummer]
    END;

  WriteLn;
  WriteLn ( 'AUGEN NACH UNTEN !' : Haelfte );
  WriteLn;
  WriteLn ( 'Jetzt die Nummern aufrufen : ' );
  Aufruf := Leer;

REPEAT
  write ( '? ' : Haelfte );
  ReadLn ( Nummer );
  Aufruf := Aufruf + [Nummer] ;
  Haus := Karte — Aufruf = Leer

UNTIL Haus;

  WriteLn;
  WriteLn ( 'Glückwunsch !' : Haelfte );
  WriteLn ( 'Ihre Nummern sind :' );
  WriteLn;

  FOR Nummer := 1 TO 90 DO
    IF Nummer IN Karte THEN
      write ( Nummer : Spalten DIV 8 )

END.
```



Der Record-Befehl

In diesem Artikel gehen wir nochmals auf Sets ein und untersuchen dann den Datentyp „Record". Mit Record lassen sich unterschiedlich strukturierte Daten bequem zusammenfassen.

In der letzten Folge wurde gezeigt, wie sich Programmabläufe durch den Einsatz von Sets vereinfachen lassen. Element eines Sets kann jeder echte Skalartyp (kein Real) sein. Außer dem Operator IN, der testet, ob ein Objekt Element eines Sets ist, werden für Sets die normalen Operatoren eingesetzt. Zwar gibt es keinen Mechanismus, der ein bestimmtes Element aus einem Set herausnimmt, doch läßt sich mit den Operatoren <= und >= feststellen, ob ein Set Teilmenge eines anderen Sets ist, während = und <> die Sets auf Mengengleichheit testet.

Für mathematische Anwendungen dürfen Mengen nicht nur unbegrenzte Objekte enthalten, sondern können theoretisch als unendlich definiert werden. (Das ist der Vorteil, wenn Probleme nur auf dem Papier gelöst und nicht in funktionierende und physisch begrenzte Hardware umgesetzt werden müssen.) In einem Computer sind Sets daher in Bezug auf Größe und Geltungsbereich Grenzen gesetzt. Typen wie:

```
RiesenSet = SET OF Ganzzahlen;
```

sollten deshalb nicht definiert werden. Ein Set-Element an der Untergrenze eines Sets muß immer einen Ordinalwert von Null oder darüber haben, während die absolute Obergrenze je nach Computer zwischen 255 und 4095 liegt. Beachten Sie, daß die Leermenge (durch [] dargestellt) Element aller nur möglichen Sets ist, unabhängig von deren Typ. Obwohl dies nicht der exakten Typendefinition von PASCAL zu entsprechen scheint, wird in der Praxis der Typ der Leermenge immer von den Typen der anderen Sets eines Ausdrucks bestimmt.

Sets als Testabfragen

Es ist nicht möglich, Untersets in Sets des gleichen Typs direkt einzuschließen. Außerdem sind die Operatoren < und > für Sets nicht definiert. Der Grund dafür liegt in der praktischen Anwendung – die meisten Einschränkungen von PASCAL ergeben Sinn, wenn sie vom Standpunkt des praktischen Einsatzes oder der reinen Logik her betrachtet werden. Die Beantwortung der Frage, ob ein Set echte Teilmenge eines anderen Sets ist, muß daher mit einem doppelten Test erfolgen:

```
( A >= B ) AND ( A <> B )
```

Bei dem Test von „Haus" im Bingo-Programm der letzten Folge war dieser spezielle Test nicht nötig, da der Set der aufgerufenen Zahlen nur größer oder gleich dem Set der in der Karte enthaltenen Nummern sein konnte, wobei der erste Fall weitaus wahrscheinlicher ist. Der folgende boolesche Ausdruck

```
Haus := Aufruf >= Karte
```

ergibt „true", wenn alle Elemente von „Karte" in dem Set „Aufruf" enthalten sind. Durch diese Eigenschaften und Funktionen eignen sich Sets ideal als Datenstrukturen für individuelle Problemlösungen. Eins der praktischsten Einsatzgebiete von Sets sind Tests von Untersets des Typs „char". Mit dem folgenden Programm lassen sich Spiele steuern:

```
Nein := ['N','n'] ;
Ja   := ['J','j'];

REPEAT
  (* Spielablauf und Anzeige *)
  (* der Gewinnpunkte        *)
  write ( 'Noch ein Spiel ? ');
  ReadLn (Antwort),

WHILE NOT ( Antwort IN Ja + Nein ) DO
  BEGIN
    WriteLn ( 'J(a) oder N(ein)' : Spalte );
    write ( 'Noch ein Spiel ? ');
    ReadLn ( Antwort );
  End

UNTIL Antwort IN Nein
```

Wenn Sie auf die Elemente einer Datenstruktur einzeln zugreifen möchten, haben Sie die Wahl zwischen Arrays, Files und Records. Besonders die Records lassen sich für praktische Anwendungen sehr gut einsetzen, da sie unterschiedliche Datentypen speichern können.

In kommerziellen Programmen gibt es oft Datensätze, deren Felder Namen, Adressen, Telefonnummern, Rechnungsnummern etc. enthalten. Die eingesetzte Datenstruktur muß sich dabei als Einheit verarbeiten lassen, aber auch den Zugriff auf die einzelnen Felder erlauben. In PASCAL können Informationen dieser Art als Einheit zusammengefaßt und verarbeitet werden, wobei die einzelnen Felder individuell abrufbar sind und gemäß ihrem Datentyp bearbeitet werden können. Gemischte Records lassen sich leicht anlegen:

```
TYPE
  Raum = RECORD
       Nummer : 1..999;
       Richtung : ( Nord, Ost, Sued, West );
       Besetzt  : boolean
    END; (* Raum *)
```

```
VAR
    Buero : Raum;
```

Ebenso wie CASE durch den Einsatz des reservierten Wortes END eine Sonderstellung einnimmt, so ist auch die Definition eines Records im Deklarationsteil die Ausnahme zu der Regel, daß BEGIN und END nur paarweise eingesetzt werden dürfen. END sollte daher zur leichteren Identifizierung immer den Namen des Records als Kommentar enthalten. Jede Variable des Typs „Raum" ist nun aus drei verschiedenen Feldern zusammengesetzt. In diesem Fall ist jedes Feld ein anderer Skalartyp. Es könnte aber auch der gleiche Typ – einfach oder strukturiert – sein. Es gibt keine Einschränkung, welche Typen in einem Record untergebracht werden dürfen. Ein Feld kann daher durchaus ein Array von Files enthalten, in denen Records untergebracht sind. Diese Records können wiederum aus Sets bestehen.

Die WITH-Anweisung

Zwischen den Ausdrücken RECORD und END gleicht die Syntax exakt der VAR-Deklaration. Dabei werden jedoch Feldnamen deklariert, die Teil der Datenstruktur sind. Die Wörter „Nummer", „Richtung" und „besetzt" gibt es daher außerhalb des „Bereichs" des Record-Namens nicht. Sie sind lokale Variablen und können daher in weiteren Teilen des Programms in anderen Zusammenhängen eingesetzt werden. Es gibt zwei Möglichkeiten, auf die Felder zuzugreifen: mit dem Punkt (.) und der Anweisung WITH.

Zur Wahl eines bestimmten Feldes wird der Name des gesamten Records mit einem Punkt von dem Feldnamen getrennt.

```
Buero. Nummer
```

bezieht sich nur auf die Ganzzahl des Feldes „Nummer". Der Inhalt von Record kann folgendermaßen initialisiert werden:

```
read ( Buero.Nummer );
'Buero.Richtung := Ost;'
'Buero.besetzt := Bewohner <> [ ]'
```

Beachten Sie, daß im letzten Beispiel die Leermenge „Bewohner" eingesetzt wurde, deren Typ nicht deklariert ist.

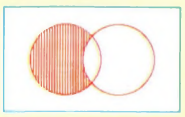
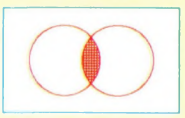
Wenn auf alle oder fast alle Felder eines Records zugegriffen werden soll, kann eine Zuweisung mit Punkten recht umständlich werden. Mit WITH ist dies weitaus einfacher. WITH bedeutet: „Ich möchte diesen Record bearbeiten und gebe daher nur die Feldnamen an". Der Aufbau der WITH-Anweisung entspricht dem der WHILE-Schleife. Die oben aufgeführte Initialisierung eines Records sieht mit WITH weitaus klarer aus:

```
WITH Buero DO
  BEGIN
    Nummer := 123;
    Richtung := Ost;
    Besetzt  := true;
  END
```

PASCAL-Operatoren für Sets

Diese Liste enthält alle Set-Operationen, die in PASCAL möglich sind (mit ihren Venn-Diagrammen).
1) Vereinigungsmenge (S1 + S2): Die Vereinigungsmenge zweier Mengen enthält alle Elemente, die in S1 und in S2 enthalten sind. (Diagramm 1)
2) Mengendifferenz (S1 – S2): Die Mengendifferenz ist die Untermenge aller Elemente von S1, die nicht zu S2 gehören. (Diagramm 2)
3) Durchschnittsmenge (S1 * S2): Die Durchschnittsmenge enthält alle Elemente, die in beiden Sets gleichzeitig vorkommen. (Diagramm 3)
4) Mengengleicheit (S1 = S2): Mengengleichheit entsteht, wenn die Menge S1 mit der Menge S2 identisch ist. (Diagramm 4)
5) Mengenungleichheit (S1 <> S2): Ergibt „true", wenn nicht jedes Element von S1 auch Element von S2 ist. (Diagramm 5)
6) Teilmenge (S1 <= S2): Ergibt „true", wenn jedes Element von S1 ebenfalls Element von S2 ist.
7) Teilmenge (S1 >= S2): Ergibt „true", wenn jedes Element von S2 ebenfalls Element von S1 ist.
8) Enthalten in (E IN S1): Ergibt „true", wenn der Set mit einem Element ([E]) ein Unterset von S1 ist.

1

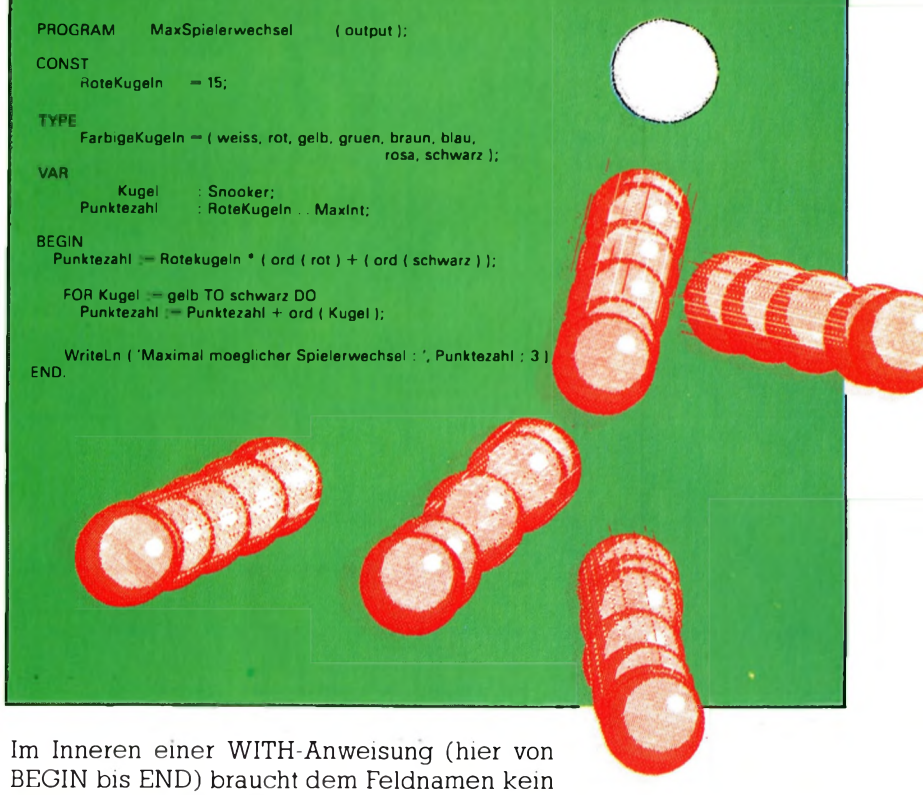
2

3

4

5

Billard

Hier ist die Auflösung für die Aufgabe der letzten Folge. Es sollte ein Programm geschrieben werden, das die maximal möglichen Spielerwechsel in einem Billardspiel berechnet.

```
PROGRAM     MaxSpielerwechsel     ( output );

CONST
      RoteKugeln     = 15;

TYPE
      FarbigeKugeln = ( weiss, rot, gelb, gruen, braun, blau,
                                                rosa, schwarz );
VAR
            Kugel       : Snooker;
      Punktezahl        : RoteKugeln . . MaxInt;

BEGIN
  Punktezahl := Rotekugeln * ( ord ( rot ) + ( ord ( schwarz ) );

    FOR Kugel := gelb TO schwarz DO
      Punktezahl := Punktezahl + ord ( Kugel );

    WriteLn ( 'Maximal moeglicher Spielerwechsel : ', Punktezahl : 3 )
END.
```



Im Inneren einer WITH-Anweisung (hier von BEGIN bis END) braucht dem Feldnamen kein Record-Name und Punkt voranzustehen. Auch wenn eine andere Variable des Programms ebenfalls den Namen „Nummer" hat, entsteht kein Konflikt, da lokale Variablen immer Vorrang haben. Die Feldbezeichnung mit Punkt ist nicht an diese Grenzen gebunden:

```
Buero.Nummer := nummer
```

ordnet beispielsweise die externe Variable (Nummer) dem Feld (nummer) des Datensatzes zu. Da PASCAL keine Unterscheidung zwischen Groß- und Kleinschreibung vornimmt, muß sich der nicht näher definierte Variablenname „nummer" auf eine externe Variable beziehen (sofern er nicht in eine WITH-Anweisung eingeschlossen ist). Eine Zuordnung zwischen Variablen des gleichen Record-Typs muß jedoch mit der Punktschreibung erfolgen:

```
Fassade.Richtung := Aussicht.Richtung
```

Über WITH könnte hier nur das Feld eines Records angesprochen werden, da sonst eine Mehrdeutigkeit auftritt, die der Compiler nicht zuläßt. Möglich wäre aber auch:

```
WITH Fassade DO
    Richtung := Aussicht.Richtung
```

Beide Schreibformen haben ihren speziellen Anwendungsbereich, der sich aus den Programmanforderungen ergibt.

In den nächsten Folgen werden wir Arrays und Files genauer untersuchen und an Beispielen zeigen, wie die Möglichkeiten dieser flexiblen Strukturen genutzt werden können.

Von Yards, Feet und Inches

Das Programm „Strecken" verarbeitet zwei Längeneinheiten, die per Tastatur in Yards, Feet und Inches eingegeben werden. Über die Definition eines Record-Typs wird jedem Wert der drei unterschiedlichen Maßeinheiten ein eigenes Feld zugeordnet. Das Programm addiert dann die beiden Strecken, wobei mit DIV und MOD festgestellt wird, ob ein Übertrag vom Feld „Inches" in Feld „Feet" und von „Feet" in „Yards" nötig ist. Die Ergebnisse werden den Feldern von „Gesamt" zugeordnet. Die Zuordnung ganzer Records kann direkt ausgeführt werden, die Bearbeitung muß jedoch für jedes Feld einzeln erfolgen. So ist „Gesamt := AStrecke + BStrecke" nicht möglich. Beachten Sie den Gebrauch des Punktes und der WITH-Anweisung beim Lesen der Werte von AStrecke und BStrecke.

```
PROGRAM         Strecken        ( input, output );

CONST
        MaxByte = 255;

TYPE
        byte    = 0 .. MaxByte;

        Strecke = RECORD
                        ins     : 0 .. 11;
                        ft      : 0 .. 2;
                        yds     : byte;

                END; (* Strecke *)
VAR
        Inches,
        Feet    : byte;
        AStrecke,
        BStrecke,
        Gesamt  : Strecke;

BEGIN

    WriteLn ( 'Strecke in Yards, Feet und Inches getrennt. );
    WriteLn ( 'durch Leerzeichen oder RETURN eingeben.' );
    WriteLn;
    write ( 'Strecke A : ' : 15 );
    read ( AStrecke.yds, AStrecke.ft, AStrecke.ins );
write ( 'Strecke B : ' : 15 );

    WITH BStrecke DO
      read (yds, ft, ins );

    Inches := AStrecke.ins + BStrecke.ins;
    Feet := AStrecke.ft + BStrecke.ft + Inches DIV 12;
    WriteLn;
    WITH Gesamt DO
      BEGIN
        ins := Inches MOD 12;
        ft := Feet MOD 3;
        yds := AStrecke.yds + BStrecke.yds
                            + Feet DIV 3;
        WriteLn ( ' Die Gesamtstrecke ist :' );
        WriteLn ( yds : 25, 'yds. ', ft : 1,
                            ' ft. und ', ins : 1, ' ins.' )
      END

END.
```

Datenschutz

In unserem Bingo-Programm sollte mit Hilfe von Schleifen verhindert werden, daß bereits aufgerufene Nummern wiederholt werden. Außerdem sollte die Eingabe von illegalen Nummern ausgeschlossen werden. Die beste Methode, das Programm vor der Eingabe falscher Daten zu schützen, ist die Verwendung von WHILE-Schleifen, die auf die beiden ReadLn-Anweisungen folgen. In beiden Fällen wird die gleiche Struktur eingesetzt:

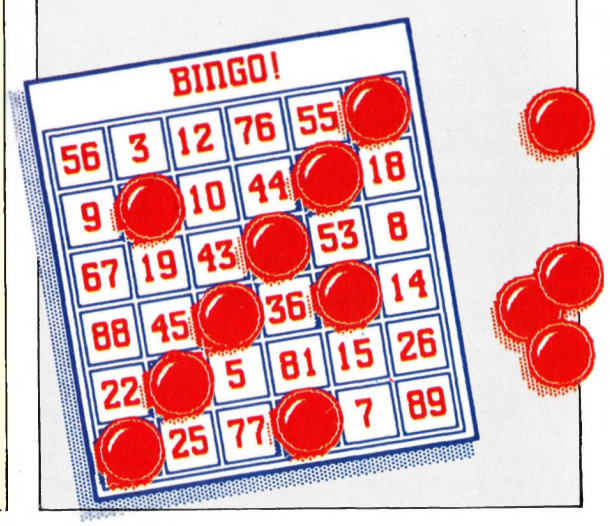
```
WHILE illegale Nummer DO
  eine Fehlermeldung ausgegeben
  einen neuen Prompt anzeigen
  die Daten nochmals einlesen (read)
```

Beim Festlegen der Kartennummern muß sichergestellt werden, daß die eingegebenen Zahlen nicht außerhalb des möglichen Bereiches (Eins bis 90) liegen und daß eine Wiederholung bereits eingegebener Nummern ausgeschlossen ist:

```
WHILE NOT ( Nummer IN moeglich ) OR
          ( Nummer IN Karte ) DO
BEGIN
  WriteLn ( Nummer : 20, ' ist nicht legal' );
  write ( 'Neueingabe: ' : 14 );
  ReadLn ( Nummer )
END;
  (* etc. *)
```

Beim Aufruf der Nummern wird die gleiche Methode eingesetzt, die Eintrittsbedingung für die Prüfschleife ist jedoch geändert:

```
WHILE NOT ( Nummer IN moeglich ) OR
          ( Nummer IN Aufruf ) DO
(* etc. *)
```



Arrays ohne Grenzen

In dieser Folge untersuchen wir, für welche Aufgaben sich PASCAL-Arrays am besten eignen, wie sie „gepackt" und indiziert werden, und wieviele Dimensionen sie haben können.

In vielen Programmiersprachen lassen sich reale Daten – Listen, Tabellen, Matrizen etc – am einfachsten mit Arrays darstellen. Da Arrays außerdem leicht auf einzelne Elemente zugreifen können, werden sie häufig verwendet. Hinzu kommt, daß die frühen Programmiersprachen nur diese eine Methode hatten, um Daten auf Computerebene einigermaßen übersichtlich anordnen zu können.

In diesem Kurs haben wir jedoch erfahren, wie PASCAL durch bedingungsgesteuerte Schleifen (WHILE und REPEAT) zusätzliche Flexibilität gewinnt und wie leicht sich Datenstrukturen wie Sets und Records für komplizierte Aufgaben einsetzen lassen. Daher spielen Arrays in PASCAL nicht die universelle Rolle, die sie in anderen Programmiersprachen einnehmen.

Die meisten Programmierer werden in den PASCAL-Arrays viele vertraute Strukturen wiedererkennen. Es gibt dabei jedoch zwei wichtige Punkte zu beachten. PASCALs Datenstrukturen können Programmieraufgaben oft besser und eleganter lösen als Arrays. Arrays sind in PASCAL jedoch nur wenigen Einschränkungen unterworfen. Sie lassen sich daher weitaus flexibler einsetzen als in vielen anderen Programmiersprachen.

Die Definition eines Array-Typs reserviert einen bestimmten Speicherbereich für das Array und legt den Typ der Array-Elemente und den Basistyp des Indices fest. So reserviert

```
TYPE
  CharZahl = ARRAY ['A'..'Z'] OF integer;
VAR
  Liste : CharZahl;
```

einen Speicher für 26 Integer, die über den Array-Namen gefolgt von einer – in eckige Klammern gestellten – Indexangabe angesprochen werden können. Arrays werden in PASCAL immer mit eckigen Klammern angegeben. Ursprünglich galt diese Schreibweise auch für BASIC; da einige Zeichensätze jedoch nur runde Klammern zur Verfügung hatten, entschied man sich für die runden Klammern. Ein Integer der oben definierten Liste läßt sich folgendermaßen ansprechen:

Liste ['M'] oder Liste [pred (Symbol)]

Im zweiten Beispiel muß der Ausdruck „pred" (Symbol) natürlich einem „char"-Wert im Unterbereich von „A" bis „Z" entsprechen. Der Index einer Array-Dimension kann jeder echte Skalartyp sein. Reals oder strukturierte Typen sind jedoch nicht zugelassen, damit Eingaben wie Liste ['Zweiten'] oder Flag [3.75] vermieden werden. Die einzelnen Elemente des Array-Typs „Char-Zahl" (wie oben definiert) lassen sich folgendermaßen mit dem Wert Null initialisieren:

```
VAR
  Buchstabe : char;
  Zaehler   : CharZahl;
BEGIN
  FOR Buchstabe := 'A' TO 'Z' DO
    Zaehler [ Buchstabe ] := 0
```

Zaehler [1] wäre in dieser Struktur natürlich illegal (falscher Indextyp) und auch Zaehler ['a'] ist nicht möglich (der Index liegt außerhalb des definierten Unterbereiches). Um Fehler zu vermeiden, sollten Sie daher für alle Indexvariablen entsprechende Typennamen definieren. Beim Schreiben von eigenen Prozeduren und Funktionen ist dies sowieso kaum zu vermeiden.

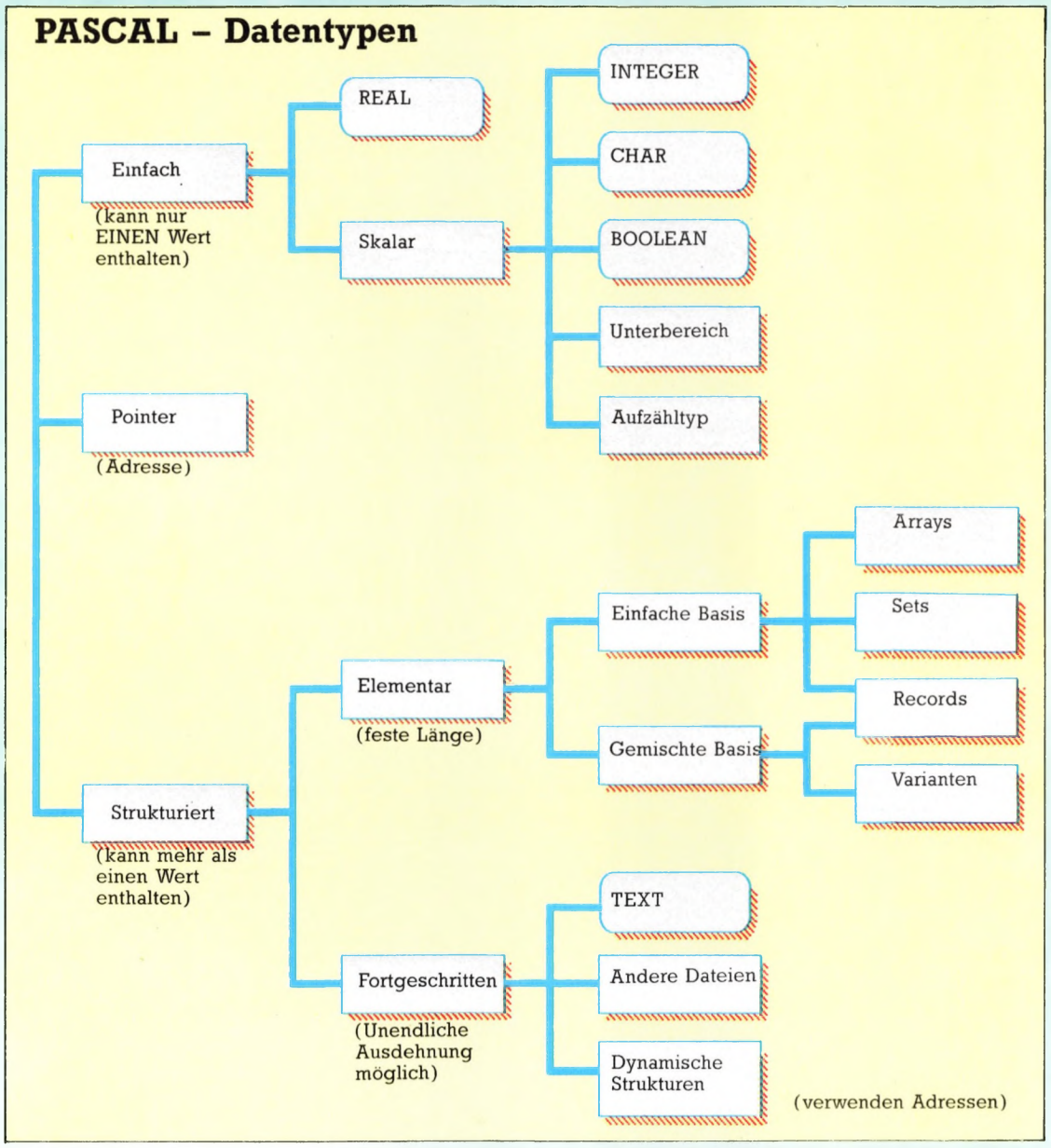
Der String-Typ

Wenn Unterbereiche von Werten bestimmt werden, die über CONST definiert wurden, können ganze Programme oft durch die Änderung einer einzigen Zeile auf andere Größen umgestellt werden. So ließe sich der Typ „String" (der in PASCAL nicht vordefiniert ist) auf folgende Weise einrichten:

```
CONST
  StringLaenge = 25;
TYPE
  StringGroesse = 1..StringLaenge;
  String = PACKED ARRAY [StringGroesse]
    OF char;
```

Beachten Sie, daß das reservierte Wort „PACKED" jedem reservierten Wort der Strukturtypen (SET, ARRAY, RECORD oder FILE) vorangehen darf. Durch das „Packen" von Datenstrukturen läßt sich die Speicherplatzbelegung

Die disziplinierte Datenstruktur von PASCAL zwingt den Programmierer, beim Aufbau seiner Programme sehr systematisch vorzugehen. Nur einer der hier gezeigten drei Grundtypen läßt sich nicht in weitere Kategorien unterteilen. Es muß daher für alle Daten eines Programms exakt definiert werden, zu welchem Datentyp sie gehören. Dieser zusätzliche Aufwand schafft jedoch die Voraussetzungen für elegante Lösungen, die in vielen anderen Programmiersprachen nicht möglich sind.



bedeutend verringern.

Beim Packen von Daten müssen Sie nur die String-Konstanten besonders beachten. Ein in Anführungszeichen eingeschlossener String wird von Eins (nicht von Null) an indiziert und muß daher ausdrücklich als

PACKED ARRAY [1..N] OF char

deklariert werden (N ist die Anzahl „char" innerhalb des Strings). Sehen Sie sich folgendes Programm an:

```
PROGRAM PackString;
CONST
  Pascal = 'Pascal';
TYPE
  String  = PACKED ARRAY [ 1..10 ] OF char;
VAR
  S       : String;
Begin
  S := PASCAL;
  S := 'Zu viele Zeichen';
  S := 'Pascal
END.
```

In diesem Programm sind die ersten beiden Anweisungen illegal, da die zugewiesenen Strings nicht die geforderte Länge haben. Die dritte Anweisung funktioniert, da der String mit vier Leerzeichen auf die erforderliche Länge gebracht wurde.

Bildschirmausgabe

Wäre String nicht als „PACKED" definiert, ließe sich die Zeichenkette nicht zuordnen. Obwohl PASCAL „read" und „write" im Normalfall nur dann für ganze Strukturen zuläßt, wenn diese auf ein Speichermedium abgelegt oder von dort eingelesen werden sollen, können die Zeichenstrings oder Stringvariablen dieses Typs dennoch direkt auf den Bildschirm ausgegeben werden. Ändern Sie zur Übung das Programm so um, daß es folgende Anweisungen unterstützt:

```
REPEAT
    write ( 'Eingabe String (Q = Ende): ' ),
```

```
    ReadLn (S),
    WriteLn ( 'Der String ist :"',S,'"')
  UNTIL S [1] IN ['Q','q']
```

Testen Sie mit dem laufenden Programm dann folgende Punkte:

- Werden vorangehende Leerzeichen oder TABS ignoriert?
- Was passiert, wenn die Zeile länger ist als die Länge des Strings?
- Was enthält S, wenn nur ein Zeichen eingegeben wurde?
- Was passiert, wenn nur RETURN gedrückt wurde?
- Können Sie Elemente, die zu wenig Zeichen enthalten, mit Leerzeichen auffüllen?

Speichergrenzen

In PASCAL können Arrays beliebig viele Dimensionen mit beliebig vielen Elementen enthalten. Folgende Typendefinitionen mögen Computer allerdings überhaupt nicht.

```
  RiesenTyp = ARRAY [integer] OF SET OF char,
    (* mindestens 64K × 128 Bytes!*)
  NochGroesser = ARRAY [ 1 .1000 ] OF
    RECORD
      Nachname,
      Vorname : string;
      Adresse : ARRAY [1 .5] OF String;
      Teilname : SET OF Anwesend;
      (* etc. *)
    END: (NochGroesser *)
```

Hier hat der Arbeitsspeicher des Computers den vom Programm angeforderten Bereich natürlich nicht mehr zur Verfügung.

In den nächsten Folgen werden wir uns mit dem Aufbau und dem Einsatz von Files beschäftigen. Da sich bei PASCAL nur jeweils ein oder zwei Datensätze im Arbeitsspeicher zu befinden brauchen, entfallen viele Beschränkungen die durch die Größe des verfügbaren Speichers bedingt sind.

Das Sieb des Eratosthenes

Dieses bekannte Primzahlenprogramm wird oft für Geschwindigkeitstests eingesetzt. Die Ausführzeit ist optimiert, da alle geraden Zahlen ignoriert und ein Array mit Flags (8192 Elemente für Primzahlen bis 16 384) eingesetzt wurde. Das Programm belegt jedoch mindestens acht KByte, in Programmiersprachen ohne boolesche Variablen (Ein-Byte) und Integer (Vier-Byte) sogar bis zu 32K. Da wir für die PASCAL-Version jedoch nur 16 384 Bit (2K) benötigen, können wir den ursprünglichen Algorithmus von Eratosthenes exakt umsetzen:

Alle Zahlen (1. .Maximum) befinden sich in einem Sieb.
Die Zahl Eins wird herausgenommen und (falls gewünscht) angezeigt.
REPEAT
  Die kleinste Zahl wird aus dem Sieb herausgenommen und angezeigt.
  Alle Vielfache dieser Primzahl werden aus dem Sieb herausgenommen.
UNTIL das Sieb leer ist.

Ein Set dieser Größe ist nur auf Großcomputern möglich, ein Array mit kleineren Sets kann den Ablauf jedoch simulieren. Der Array-Index jedes Sets wird über die Ganzzahlenteilung der bearbeiteten Zahl gefunden, wobei ihre Eigenschaft als Set-Element von N modulo 100 oder 1000 dargestellt ist. Diese Programmversion ist langsamer als die Array-Version.

```
PROGRAMM      EratosthenesSieb      ( output );
(* Primzahlengenerierung durch den Algorithmus des
Eratosthenes *)

CONST
  SetGroesse     =   100; (* vom Compiler abhaengig *)
  PredSetGroesse =    99; (* SetGroesse - 1 *)
  MaxPrim        = 16383; (* fuer MaxInt = 32767 *)
  ListMax        =   163; (* MaxPrim DIV SetGroesse *)

TYPE
  PrimBereich    = 1 .. MaxPrim;
  Dimension      = 0 .. ListMax;
  SetBereich     = 0 .. PredSetGroesse;
  Si             = SET OF SetBereich;
  Eratosthenes   ARRAY [ Dimension ] OF Si;
  Kardinal       = 0 .. MaxInt;

VAR
  Sieb           : Eratosthenes;
  Zaehler,
  Vielfaches     : Kardinal;
  N              : Dimension;
  Index          : PrimBereich;
  Nummer         : 0 .. PredSetGroesse;

BEGIN
  WriteLn;
  WriteLn ( 'Sieb des Eratosthenes' : 50 );
  WriteLn ( '----------------' : 50 );
  WriteLn;
  WriteLn;

  For Index := 0 TO ListMax DO
    Sieb [ Index ] := [0 .. PredSetGroesse ];
      (* Alle Zahlen in das Sieb laden *)

  Sieb [ 0 ] := [ 2 .. PredSetGroesse ];
(* WriteLn ( 1 ); *) (* Primzahl per Definition *)
  Zaehler := 1;

  FOR N := 2 TO MaxPrim DO
    BEGIN
      Index := N DIV SetGroesse;
      Nummer := N MOD SetGroesse;

      IF Nummer IN Sieb [ Index ] THEN
        BEGIN
          Zaehler := succ (Zaehler );
        (* WriteLn ( N ); *)
          Sieb [ Index ] := Sieb [ Index ] - [ Nummer ];

          Vielfaches := N + N;

          WHILE Vielfaches <= MaxPrim DO
            BEGIN
              Index := Vielfaches DIV SetGroesse;
              Sieb [ Index ] := Sieb [ Index ] -
                  [ Vielfaches MOD SetGroesse ];
              Vielfaches := Vielfaches + N
            END
        END
    END;

  WriteLn;
  WriteLn (Zaehler : 25, ' Primzahlen gefunden.' )

END.
```

Eigene Abläufe

In PASCAL können Sie von der Sprache vorgegebene Prozeduren und Funktionen durch eigene Abläufe ergänzen. In diesem Artikel untersuchen wir, welche Unterschiede zwischen Funktionen und Prozeduren bestehen, und gehen auf ihre Gültigkeitsbereiche ein.

In PASCAL können Sie eigene Funktionen und Prozeduren definieren, wenn Ihnen die von der Sprache vorgegebenen Möglichkeiten wie „write" oder „sqrt" nicht genügen. Bevor wir uns jedoch diesen Mechanismen zuwenden, wollen wir zunächst untersuchen, warum beispielsweise „write" eine Prozedur ist, aber keine Funktion, und „sqrt" eine Funktion, aber keine Prozedur. Bei dem Befehl

```
write ('Hallo!')
```

erwarten wir, daß die als Parameter angegebene Zeichenfolge in dieser Reihenfolge ausgegeben wird. Mit anderen Worten: „write" ruft ein Unterprogramm auf, das die Daten als Zeichenfolge auf den Bildschirm bringt. Was aber passiert bei folgender Anweisung:

```
sqrt (256)
```

Ein Compilerfehler wird angezeigt – Anweisungen dieser Art sind nicht legal. Der Befehl

```
write(sqrt (256))
```

wäre jedoch völlig in Ordnung: Auf dem Bildschirm erscheint 1.60000E+01.

Da die write-Anweisung einen Ablauf auslöst (hier: Daten an ein Ausgabegerät sendet) wird sie als Prozedur per Namen aufgerufen. Eine weitere von PASCAL vorgegebene Prozedur ist beispielsweise „page(F)", (in der Textdatei F wird eine neue Seite angesprochen). Der Ausdruck „sqrt" dagegen liefert als Ergebnis nur eine reale Zahl – die Quadratwurzel des Argumentes. Er muß immer mit einem einfachen numerischen „Argument" versehen sein. Im Gegensatz zu Prozeduren sind Funktionen daher keine Anweisungen: Ebenso wie die Zahl 16 ohne weitere Information nur diesen Wert darstellt, haben Ausdrücke wie „sqrt(X/3)" oder „odd(N)" ebenfalls keine eigenständige Bedeutung. Die Namen der Funktionen können daher wie Variablen angesehen werden, die zwar nie initialisiert wurden, aber automatisch berechnet werden, wenn ihr Name im Programm auftaucht. Der Ergebniswert wird dabei immer vom aktuellen Wert des Argumentes abgeleitet, das heißt, er ist Funktion des Argumentes.

Prozeduren liefern jedoch keine Ergebniswerte und können daher auch nicht in Ausdrücke eingesetzt werden. Ein Befehl wie:

```
N:=WriteLn
```

ist ebensowenig möglich wie LET N = PRINT in BASIC.

PASCAL erleichtert das Schreiben umfangreicher Programme, indem es die Möglichkeit bietet, eigene Funktionen und Prozeduren anzulegen, deren Zugriff und Verbindung untereinander exakt gesteuert ist. Da die Namen hier wie bei den Variablenbezeichnungen nur wenigen Einschränkungen unterworfen sind, können Sie für Ihre eigenen Unterprogramme eindeutige und ausführliche Bezeichnungen angeben. Sehen Sie sich folgende Programmstruktur an:

```
PROGRAM Beispiel (input, output, Datei);
                        (* Deklarationen . . *)
BEGIN
  Oeffne (Datei);
  WHILE NOT EoF (Datei) DO
    BEGIN
      read (Datei, Inhalt);
      Bearbeite (Inhalt)
    END
END.
```

EoF und EoLn

Obwohl wir Dateizugriffe noch nicht behandelt haben, sollten Sie den Inhalt des Programms problemlos verstehen können. Im Deklarationsteil des Programms wäre dabei eine Prozedur aufgeführt, die auf dem Speichermedium eine Datei sucht und sie zum Lesen eröffnet (Oeffne). Eine weitere Prozedur verarbeitet dann die Datei (Bearbeite). Das Programm enthält die von PASCAL ebenfalls vordefinierte Prozedur „read", die wir bisher nur zum Einlesen von Texteingaben eingesetzt haben, die aber ebenfalls strukturierte oder unstrukturierte Daten aus entsprechenden Dateien lesen kann. Die boolesche Funktion „EoF" (End of File – Dateiende) liefert den Wert „true", wenn der letzte Datensatz der als Argument angegebenen Datei gelesen wurde. Wird der Dateiname weggelassen, dann nimmt die Funktion die Standarddatei als Argument an. Bei Textdateien liefert die Funktion „EoLn" (End of Line – Zeilenende) den Wert „true", wenn das letzte Zeichen einer Zeile gelesen wurde.

Abgesehen von den oben genannten Unterschieden haben Programme, Prozeduren und Funktionen vieles gemeinsam – alle sind Module mit eigenen lokalen Variablenbeschreibungen und Anweisungsfolgen. So ist der Inhalt des Kopfes fast der einzige wesentliche

Syntax-Unterschied. Ein Programmkopf besteht aus dem reservierten Wort „PROGRAM", einem vom Anwender vorgegebenen Namen und einer Parameter-Liste, die die Dateien angibt, mit denen das Programm Daten austauscht. Ein Prozedurkopf dagegen besteht aus dem reservierten Wort „PROCEDURE" und einer Parameter-Liste mit den Typennamen der Datenvariablen. Hier ein Beispiel:

```
PROCEDURE Zeilen (Anzahl Zeilen : integer);
  VAR
    N : integer;
  FOR N := 1 TO AnzahlZeilen DO
      WriteLn
END;
```

Der END-Anweisung einer Prozedur folgt immer ein Semikolon und kein Punkt wie am Ende eines Programms. Dem formalen Parameter-Namen „AnzahlZeilen" wird beim Aufruf der Prozedur der jeweils aktuelle Wert übergeben:

```
Zeilen (5)
```

Zugegeben, der Vorgang ist recht simpel, wenn Sie aber oft mehrere Leerzeilen nacheinander ausgeben müssen – in diesem Falle fünf – dann erspart Ihnen die Prozedur die lästigen Folgen von WriteLn-Anweisungen. Mit dieser Allzweckroutine können Sie soviel Leerzeilen ausgeben, wie Sie jeweils brauchen. „Zeilen (10)" erzeugt 10 Leerzeilen, „Zeilen (20)" 20 etc.

Unsere einfache Prozedur ist ebenfalls ein gutes Beispiel für die Datensicherheit, die PASCAL bietet. So muß die Steuervariable für die FOR-Schleife als echte lokale Variable deklariert sein. Es ist nicht möglich

```
FOR AnzahlZeilen := 1 To AnzahlZeilen DO . .
```

zu schreiben, da die Sicherheit einer Prozedur nicht garantiert werden kann, wenn die Steuervariable nicht lokal oder auch nur „relativ global" ist. Haben Sie schon einmal stundenlang nach einem Fehler im BASIC-Programm gesucht, das Programmzeilen dieser Art enthielt

```
300 FOR N = 1 TO T
400 GOSUB 2000
500 NEXT N
```

um schließlich herauszufinden, daß N von der Unterroutine mit einem anderen Wert belegt wurde? PASCAL fängt Fehler dieser Art von vornherein ab und spart Ihnen so viel Zeit bei der Programmentwicklung. Jeder Name, der im Inneren eines Blocks deklariert wird, kann sich nur auf diesen Block beziehen. Sein Gültigkeitsbereich (das heißt der Teil des Blocks, von dem aus auf diesen Namen zugegriffen werden kann) erstreckt sich zwar vom Punkt seiner Deklaration bis zum Blockende, kann aber durch Neudefinition weiter eingeschränkt werden. Die Variable N des vorigen Beispiels bezog sich auf das Innere der Prozedur „Zeilen" und ist damit eine lokal deklarierte Ganzzahl. Diese Deklaration hat bei jedem Prozeduraufruf automatisch Vorrang vor globalen Variablen gleichen Namens.

Bereichsweise

```
PROGRAM Bereich;

VAR
    N : integer;
    X : real;
PROCEDURE A ( Y : real );

  TYPE
    X = SET OF char;
  (* etc. *)

PROCEDURE B ( N : integer );
  (* etc. *)
BEGIN (* Hauptprogramm —
  Bereich *)
  ...
  A ( succ (N) / 3 );
  B ( N );
  A ( X );

  (* etc. *)
END.
```

Die untenstehende Tabelle zeigt den Bezug und die Gültigkeitsbereiche der Variablen des nebenstehenden Kurzprogramms.

Prozedur/Var	Bezug	Gültigkeit
A	Hauptprog	A,B,Hauptprog
B	Hauptprog	B,Hauptprog
N(global)	Hauptprog	A,Hauptprog.
X(real)	Hauptprog	B,Hauptprog
X(in Proz A)	A	A
N(in Proz B)	B	B

Die globalen Variablen N und X im Hauptprogrammteil von „PROGRAM(Bereich)" beziehen sich auf das gesamte Programm. Der Gültigkeitsbereich von X reicht daher mit Ausnahme der Prozedur A von ihrer Deklaration bis an das Programmende. In A wird X als Typenname für „X(SET OF char)" verwendet, der aber nur für Prozedur A Gültigkeit hat. In ähnlicher Weise ist N der formale Parameter der Prozedur B und keine globale Variable.

A und B beziehen sich weiterhin zwar auf das gesamte Programm, doch fängt der Gültigkeitsbereich von B erst bei seiner Definition an. Während Sie also A durchaus in die Prozedur B einsetzen können, ist B für A unerreichbar. Hier zeigt sich die Logik von PASCAL: Kein Programm läßt sich aufrufen, bevor es geschrieben wurde. Es wird auch klar, warum alle Definitionen und Deklarationen immer in der festgelegten Reihenfolge erfolgen müssen: CONST, TYPE, VAR und dann die Definition der Prozeduren und Funktionen, geordnet nach den strukturellen Erfordernissen des Programms. Da viele PASCAL-Compiler den Quellentext nur einmal durchgehen, ist diese logische Anordnung unbedingt notwendig.

Modularer Programmaufbau

Der zusätzliche Aufwand mit lokal deklarierten Variablen zahlt sich aus. Wenn Programme alle ihre Daten als Parameter an Prozeduren übergeben, wird ein hoher Grad an modularem Aufbau erreicht. Zwar gibt es PASCAL-Programme, die Prozeduren ohne Parameter-Listen verwenden und nur globale Daten einsetzen, doch sollte diese Art der Programmierung möglichst vermieden werden. Modularer Programmaufbau hat außer erhöhter Datensicher-

heit noch den weiteren wesentlichen Vorteil, daß ein Programmiererteam an einem großen Programm arbeiten kann, ohne sich um doppelt vergebene Variablen-Namen kümmern zu müssen. Beim Aufruf von B wird der Wert der globalen Variablen N in dieser Prozedur zwar als aktueller Parameter an eine Variable gleichen Namens übergeben, doch ist N dort eine völlig andere Variable. Das zeigt folgende wichtige Punkte auf:

- Beim „Blockeintritt" wird der übergebene Wert in eine lokale Variable kopiert.
- Eine Veränderung dieses Wertes innerhalb des Blocks hat keine Auswirkungen auf die ursprüngliche globale Variable.
- Der übergebene Wert kann ein Ausdruck jedes (korrekten) Typs sein.

Natürlich muß jeder Prozeduraufruf die aktuellen Parameter enthalten, die mit der formalen Liste am Prozeduranfang übereinstimmen. Der Übergangsmechanismus kann daher als eine Art Initialisierung der formalen Parameter-Namen mit aktuellen Werten angesehen werden.

Zeilen (succ (N+gap)DIV 2)

führt innerhalb der Prozedur folgende Zuweisung aus:

AnzahlZeilen := succ(N+gap)DIV 2

Versuchen Sie einmal herauszufinden, was passiert, wenn die übergebenen Werte innerhalb der Prozedur verändert werden.

Grundwerte

Das Programm „Grundwert" zeigt, wie sich Wertparameter von Prozeduren aus einsetzen lassen. Bei Angabe einer Basis von 2 (binär) bis 16 (hexadezimal) werden per Tastatur eingebene Dezimalzahlen in der entsprechenden Schreibweise dargestellt. So würde 32767 in Hexadezimal 7FFF werden, in Binär 111111111111111 oder 77777 in Oktal.

Versuchen Sie, folgende Änderungen in das Programm einzufügen:

○ Um Negativdarstellungen zu ermöglichen, muß die Zahl in ihr Zweierkomplement umgewandelt werden. Auf Maschinenebene wird dafür die Zahl umgekehrt und Eins addiert. Finden Sie eine Methode, die diese Aufgabe in PASCAL erledigt?

○ Möchten Sie die Umwandlung andersherum ausführen? Nehmen Sie dazu das Programm als Modell für die Eingabe einer Zahl mit Basis 2 bis 16 und für die Ausgabe im Dezimalformat. Die beiden vorgeschlagenen Programmteile lassen sich auch zu einem Programm zusammenfassen.

```
PROGRAM          GrundWert          ( Input, Output );
        (* wandelt Dezimalzahlen in Zahlenformate jeder
        Basis im Bereich binär bis hexadezimal um *)

CONST
        Spalte = 79; (* Bildschirm / Drucker *)

TYPE
        Byte     = 0 . . 255;
        Cardinal = 0 . . MaxInt;

VAR
        Zahl,
        Basis : integer;
        Ende,
        legal : boolean;

(* ------------------------------------------------------- *)

PROCEDURE          SchreibeZiffer          ( Ziffer : Byte );
    (* Wert von Liste [ Zaehler ] an Ziffer übergeben *)

  BEGIN
    IF Ziffer IN [ 0 . . 9]
      THEN
        write ( Ziffer : 1 )

      ELSE (* Darstellung als A . . F *)
        CASE Ziffer OF
            10      :  write    ( 'A' );
            11      :  write    ( 'B' );
            12      :  write    ( 'C' );
            13      :  write    ( 'D' );
            14      :  write    ( 'E' );
            15      :  write    ( 'F' )

        END; (* CASE *)

  END; (* SchreibeZiffer *)
(* ------------------------------------------------------- *)
PROCEDURE          Anzeigen          ( N          : Cardinal;
                                       Basis      : Byte );
  CONST  (* Alle diese Daten sind für die
            Prozedur lokale Daten *)
        MaxZiffer = 32;
  TYPE
        Grenze = 1 . . MaxZiffer;
  VAR
        Liste    : ARRAY [ Grenze ] OF Byte;
        Index    : Grenze;
        Zaehler : Byte

  BEGIN
    Zaehler := 0;

    REPEAT
        Zaehler := succ ( Zaehler );
        Liste [ Zaehler ] := N MOD Basis;
        N := N DIV Basis

    UNTIL N = 0;
              (* Anzeige beginnt bei MSB:*)
    FOR Zaehler := Zaehler DOWNTO 1 DO
        SchreibeZiffer ( Liste [ Zaehler ] )

  END; (* Anzeigen *)
(* ------------------------------------------------------- *)
BEGIN          (* Grundwert — Hauptprogramm *)

  REPEAT
    WriteLn ( 'Geben Sie eine Zahlenbasis:' );
    WriteLn ( ' von 2 . . 16 an' : Spalte );
    WriteLn ( '(Zahlen ausserhalb = Ende)' : Spalte );
    Write ( 'Basis ? ' );
    read (Basis );
    legal := Basis IN [2 . . 16 ];

    IF legal THEN
      BEGIN
        write ( 'Zahl zum Wandeln (0 wechselt Basis) ? ' );
        read ( Zahl );
        Ende := Zahl <= 0;

        WHILE NOT Ende DO
          BEGIN
            write ( Zahl : Spalte DIV 2,
                    ' zu Basis ', Basis : 1, ' ist ' );
            Anzeigen ( Zahl, Basis );
            WriteLn;

            write ( ' Neue Zahl ? ' );
            read ( Zahl );
            Ende := Zahl <= 0;
          END
    END

  UNTIL NOT legal

END.
```

Im Einsatz

In der letzten Folge wurde gezeigt, daß PASCAL keine Funktionsanweisungen zuläßt, da Funktionen Werte berechnen und ihre Aufgaben sich von denen der Prozeduren unterscheiden. In diesem Artikel sehen wir uns an, wie sie eingesetzt werden.

Funktionen lassen sich zwar wie Prozeduren mit Namen aufrufen, können jedoch nur in Gleichungen eingesetzt werden. Während ein Prozeduraufruf den Ablauf eines Unterprogramms auslöst, liefert der Aufruf einer Funktion den Ergebniswert einer Berechnung. Dieser Wert kann ein einfacher Typ sein (beispielsweise eine reale Zahl), ein Skalar oder ein „Pointer". Der Name einer Funktion unterscheidet sich von dem einer Prozedur nur durch das Wort FUNCTION und die Typenbezeichnung für den Ergebniswert. Die Funktion „odd" beispielsweise könnte man auch selbst definieren:

```
FUNCTION Odd (Nummer : integer) : boolean;
  BEGIN
    Odd := Nummer MOD 2 > 0
  END; (* Odd *)
```

Da das Ergebnis einer Funktion in ihrem Namen gespeichert wird, muß der Funktionsname den Typ des Ergebniswertes enthalten. Funktionsnamen lassen sich daher als Variablen ansehen, die zwar nie initialisiert wurden, deren Wert jedoch berechnet wird, wenn der Name in einer Gleichung auftaucht. Funktionsnamen können nur auf der linken Seite einer Zuordnung auftreten. Wäre die gleiche Funktion nachlässig programmiert, wie:

```
IF Nummer MOD 2 > 0 THEN
  Odd := true
```

(der Fall „ELSE Odd := false" fehlt) besteht die Möglichkeit, daß das Ergebnis nicht definiert ist.

Wie bei Prozedurparametern wird bei Aufruf einer Funktion der Wert des aktuellen Parameters an lokale Variablen (hier die Ganzzahl „Nummer") übergeben. So führt folgende Anweisung

```
WriteLn ( Odd ( sqr ( N DIV 100) ) )
```

(die immer „false" ergibt) vier Abläufe aus:

1. der Ausdruck „N DIV 100" wird berechnet;
2. das Ganzzahlergebnis wird an „sqr" als aktueller Parameter übergeben;
3. das Quadrat dieses Parameters wird an „Odd" übergeben;
4. das Ergebnis wird auf boolesche Weise bewertet und an die WriteLn-Prozedur wiederum als Wertparameter übergeben.

Der Wert N wird durch die Ausführung der Anweisung natürlich nicht verändert, da ein Funktionsergebnis ist immer nur „Funktion" der übergebenen Parameterwerte sein kann. Da PASCAL nicht die eigentlichen Variablen, sondern nur Kopien der Variablenwerte an Funktionen und Prozeduren weitergibt, ist sichergestellt, daß die ursprünglichen Parameter sich nicht ändern, selbst wenn ihre Werte im Inneren einer Funktion oder Prozedur umgewandelt werden.

Globale Konstanten

Zwar kann unter PASCAL direkt auf globale Daten zurückgegriffen werden, doch empfehlen wir, davon so wenig wie möglich Gebrauch zu machen. Alle Datenübergaben bei Prozedur- oder Funktionsaufrufen sollten von Parameterlisten gesteuert werden, selbst wenn die Daten innerhalb des Unterprogrammbereichs verfügbar sind. Die einzige Ausnahme zu dieser generellen Regel sind globale Konstanten, die in PASCAL per Definition nicht verändert werden können. Beachten Sie aber, daß konstante Werte, die als Parameter übergeben werden, zu lokalen Variablen werden und daher nicht länger geschützt sind.

```
FUNCTION Klein ( Zeichen : char ) : char;
  (* wandelt als Argument uebergebene
  Grossbuchstaben in Kleinbuchstaben um *)
  CONST
    Offset = 32; (* ASCII ord('a') — ord('A' )*)
  BEGIN
    IF Zeichen IN [ 'A'..'Z']
      THEN
        Klein := chr ( ord ( Zeichen )
                 + Offset )
      ELSE
        Klein := Zeichen
  END; (* Klein *)
```

Es kann der Fall eintreten, daß Prozeduren Werte von globalen Variablen doch verändern müssen, da ihre Aufgabe sonst hinfällig wäre. Ein gutes Beispiel dafür ist die Prozedur „read". Wenn read (N) die eingegebene Ganzzahl nur an eine lokale Variable weitergeben würde, hätte das für uns wenig Wert. Wir müssen daher die Adresse des Variablenparameters (und nicht nur den Variablenwert) übergeben, damit die Prozedur sich direkt darauf beziehen kann und nicht nur eine Kopie bearbeitet. Dieser Mechanismus sollte jedoch nur mit Prozeduren eingesetzt werden.

Die Übergabe eines Variablenparameters wird in der Parameterliste der Prozedur durch

das reservierte Wort „VAR" vor dem Variablennamen angezeigt. Der Aufbau der Parameterliste ist mit der VAR-Deklaration eines Blocks identisch, allerdings brauchen im Prozedurkopf nur die Variablennamen angegeben zu werden, die von der Prozedur verändert zurückgegeben werden sollen:

```
PROCEDURE Ablauf ( VAR Zaehler :
    ZaehlerListe );
```

Um diese vielseitig einsetzbare Technik deutlich zu machen, werden wir ein Programm entwickeln, das über die Tastatur Namen einliest und zu jedem Namen einen Geldbetrag stellt, den diese Person schuldet. Zur Vereinfachung nehmen wir einen String pro Namen und Ganzzahlen als Geldbeträge. Nachdem das Programm getestet wurde, können Sie Adressen, Telefonnummern etc. hinzufügen. Als Ergebnis wollen wir eine Liste erzeugen – in alphabetischer Reihenfolge oder nach Beträgen geordnet. Als Datenstruktur bietet sich eine Liste von „Records" an, deren Felder sich später leicht erweitern lassen. Wir können dabei jeden Record einzeln ansprechen, aber auch jedes einzelne Feld als Sortierschlüssel verwenden. Zunächst die Datendefinition:

```
CONST
  StringLaenge   = 20;
  ListenLaenge   = 50;
TYPE
  Cardinal       = 0..MaxInt;
  StringGroesse  = 1..StringLaenge;
  String         = PACKED ARRAY
                   [ StringGroesse ] OF char;
  Daten          = RECORD
                     Name     : string;
                     Schulden : Cardinal;
                   (* weitere Felder moeglich *)
                   END; (* Daten *)
  Grenze         = 1..ListenLaenge;
  RecordListe    = ARRAY [Grenze] OF Daten;
```

Wir können zwar nur fünfzig Namen mit nicht mehr als 20 Zeichen eingeben, diese Angaben lassen sich jedoch leicht ändern.

Aus folgenden Gründen haben wir die Unterbereiche und Strukturen mit TYPE definiert:

1. Sicherheit – ein Index vom Typ „Grenze" kann nie die festgelegten Grenzen eines Arrays überschreiten.

2. Fehlersuche – sollte während des Testens ein Bereichsfehler auftreten, läßt sich die Ursache leicht feststellen.

3. Effektivität – lokale Variablen sparen Speicher und vermeiden überflüssige Doppelvergaben.

4. Notwendigkeit – PASCAL „besteht" darauf, daß als Parameter nur Namen und keine Bezeichnungen wie 0..255 eingegeben werden.

Als nächstes müssen wir nun die Variablen deklarieren und einen Algorithmus aufbauen. Diesen Ablauf behandeln wir in der nächsten Folge. Entwickeln Sie als Übung ihren eigenen Programmalgorithmus, und stellen Sie die entsprechenden Prozeduren zusammen.

Geprüfte Eingaben

Das Programm „Karten-Leser" ist eine sichere Methode, positive Zahlen einzulesen. Dabei werden die Daten als Zeichen eingegeben. Die Umwandlung in den entsprechenden Zahlenwert geschieht durch die Subtraktion des Ziffernwertes von der Ziffer „0". Da die Zeichen 0..9 lückenlos definiert sind, funktioniert das Programm bei allen Zeichensätzen. Beachten Sie, daß das Programm nur zwei globale Variablen besitzt.

Da die Daten für die Zeichenverarbeitung und die Funktion „Wert" nur von der Prozedur „Karte-Einlesen" benötigt werden, wurden diese Komponenten auch nur für diese Prozedur definiert.

In dem Programm gibt es nur einen einzigen möglichen Fehler – bei der Eingabe eines Wertes größer als „MaxInt" stürzt der Zuordnungsbefehl der WHILE-Schleife ab.

```
PROGRAM          Kartenleser          ( input, output );

TYPE
        Cardinal          = 0..MaxInt;

VAR
        PosNum            : Cardinal;
        InOrdnung         : boolean;


  (* ---------- Ebene 1 ---------- *)

PROCEDURE KarteEinlesen  ( VAR N   : Cardinal;
                           VAR OK  : boolean );
  CONST
        Leerzeichen       = ' ';
  TYPE
        Einfach           = 0..9;
  VAR
        Symbol            : char;
        Ziffern           : SET OF char;

  (* ---------- Ebene 2 ---------- *)

  FUNCTION Wert ( Ziffer : char )                    : Einfach;
        (* liefert den Zahlenwert einer
           Ziffer jedes Zeichensatzes *)
    BEGIN
      Wert := ord ( Zeichen ) - ord ('0' )

    END; (* Wert *)
  (* ---------- Ebene 2 ---------- *)
                  (* Zurueck zu Ebene 1 *)
BEGIN (* KarteEinlesen *)

    REPEAT
      read ( Symbol )

    UNTIL Symbol > Leerzeichen;

    Zeichen := [ '0'..'9' ];
    OK      := Symbol IN Zeichen;

    IF OK THEN
      BEGIN
        N := 0;

        WHILE Symbol IN Zeichen DO
          BEGIN (*berechnen auf Basis 10 *)
            N := 10 * N + Wert ( Symbol );
            read ( Symbol )
          END (* Begrenzungszeichen oder 'EoLn' *)
        END    (* hinter der Zahl beseitigen *)

END; (* KarteEinlesen *)
  (* ---------- Ebene 1 ---------- *)
BEGIN (* KartenLeser — Ebene 0    Hauptprogramm *)

  page ( output );
  WriteLn;
  WriteLn ( ' Dieses Programm liest positive' );
  WriteLn ( ' Ganzzahlen ( im Bereich 0..',
                                MaxInt : 1, ')' );
  WriteLn ( 'Das Programmende wird von der' );
  WriteLn ( 'Fehlererkennung gesteuert.' );
  WriteLn;
  write ( 'Geben Sie eine Zahl ein :  ' );
  KarteEinlesen ( PosNum, InOrdnung );

  WHILE InOrdnung DO
    BEGIN
      WriteLn;
      write ( 'Die eingegebene Zahl war : ' );
      WriteLn ( PosNum : 1);
      Write ( 'Zahl ? ' );
      KarteEinlesen ( PosNum, InOrdnung )

    END;

  WriteLn ( '--- FEHLER ERKANNT ---' : 40 )

END.
```

Auf- und Abbau

Wir entwickeln einen allgemeinen Algorithmus für den Aufbau unserer Datenbank und stellen mehrere Programme vor, die die strenge Programmstruktur von PASCAL beim Aufbau und Abruf von Dateien verdeutlichen.

In unserer letzten Folge begannen wir mit dem Aufbau einer Datenbank, für die wir nun einige Variablen und folgenden allgemeinen Algorithmus definieren:

```
VAR
  Liste   : RecordListe;
  Anzahl : Cardinal;
BEGIN
  Anzahl : = 0; (* aktuelle Laenge der Liste *)
  (* Daten in die Liste einlesen und
     Anzahl aktualisieren *)
  (* Datensaetze sortieren *)
  (* Datensaetze anzeigen *)
END.
```

Für die Umsetzung dieses Algorithmus in ein Programm legen wir zunächst die Prozeduren für die drei Hauptschritte an. Nachdem wir für jede Prozedur die Parameterlisten aufgestellt haben, können wir das Gerüst für die einzelnen Prozedurblöcke entwerfen. Die letzte Anweisung des Programms soll folgendermaßen lauten:

```
Print (Liste, Anzahl)
```

Für die Übergabe der Datenliste und der Datensatzzahl brauchen wir in den Prozedurkopf von „Print" nur folgende Variablen einzusetzen:

```
PROCEDURE Print (Objekte : RecordListe,
                 MaxAnzahl : Grenze);
```

Für den Augenblick werden wir dieses Modul jedoch nicht weiterentwickeln, sondern erst die anderen Grundprozeduren entwerfen. Wir müssen dabei jede einzelne mit einem geeigneten Namen versehen, entscheiden, welche Daten als Parameter übergeben werden sollen und ob ein Teil davon das Format von „VAR"-(Adreß-)Parametern haben soll.

Bisher haben wir nur Datenstrukturen mit festen Grenzen kennengelernt. Die Grenzen wurden mit TYPE-Definitionen festgelegt und so dem Compiler mitgeteilt. PASCAL bietet aber auch Datenstrukturen, deren Größe sich während der Programmausführung verändern kann (mit bestimmten Einschränkungen sogar der Typ). Diese Strukturen werden nur durch die Kapazität des Arbeitsspeichers oder des Speichermediums begrenzt.

Wie Arrays können die Elemente einer Datei jeden strukturierten oder unstrukturierten Typ annehmen – nur eine Datei von Dateien ist nicht möglich. Um eine Datei mit einer unendlichen Anzahl Datensätze aufbauen und verarbeiten zu können, müssen wir an unsere Definition des Recordtyps nur folgende Deklaration anhängen:

```
TYPE
  DateiTyp = FILE OF Daten;
VAR
  Datei      : DateiTyp;
```

Die einzelnen Datensätze können dabei Namen, Summen oder jede andere Information enthalten. Ebenso wie wir mit „read (Zeichen)" ein Zeichen aus einer Datei lesen, so können wir mit „read (Datei, Feld)" oder „write (Datei, Feld)" ganze Datensatzstrukturen als ein Datenobjekt ansprechen. Wir brauchen diese Dateien jetzt nur noch zu eröffnen, um mit den vordefinierten Prozeduren „reset" und „rewrite" darauf zugreifen zu können.

Wenn Dateien auf ein Speichermedium geschrieben werden sollen, müssen ihre Namen in der Parameterliste des Programmkopfes aufgeführt sein.

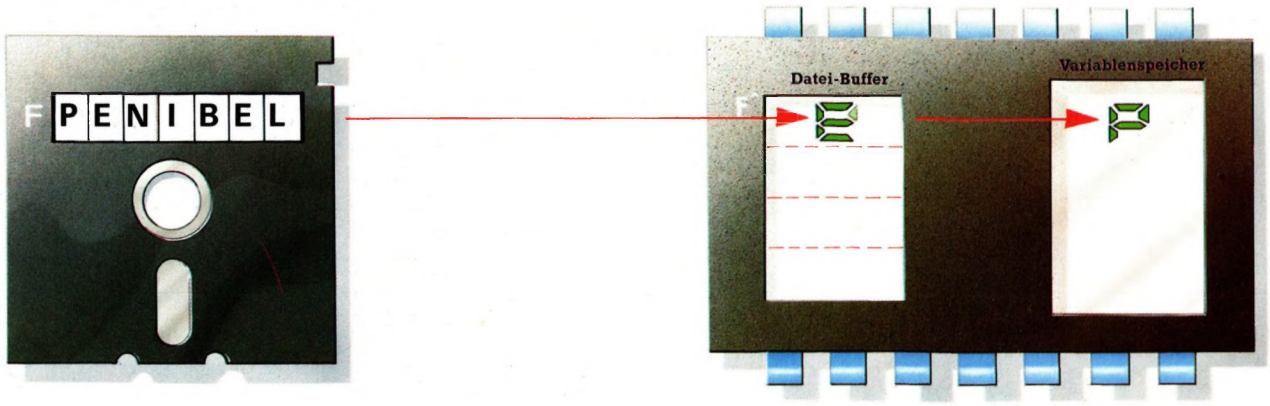
```
PROGRAM DateiVerwaltung
  (input,output,Datei);
```

Jeder Einsatz der Anweisungen „read" und „write" ruft die vordefinierten PASCAL-Prozeduren für Ein- und Ausgabe auf. Bisher kannten wir nur die Ein- und Ausgabedateien „input" und „output", die auf Microcomputersystemen normalerweise die Tastatur und den Bildschirm bezeichnen. Da PASCAL alle Systemteile als Dateien ansieht, haben alle Ein- und Ausgabevorgänge das gleiche Format. So schreibt „write (N)" ohne Angabe eines Dateinamens den Wert von N in die Datei „output", während „read" und „ReadLn" ohne Dateinamen automatisch die input-Datei ansprechen. Beides sind Textdateien – das heißt, jeder Datensatz besteht aus einem einzigen char-Wert.

Im Gegensatz zu anderen Dateiarten können Textdateien außer einer Datei-Endmarkierung noch beliebig viele Zeilen-Endmarkierungen enthalten. Diese Markierungen sind je nach Betriebssystem unterschiedlich. Sie können beispielsweise aus einzelnen Steuerzeichen oder zwei Zeichen (wie CR und LF) bestehen oder statt eines Zeichens die Zeilenlänge enthalten.

Für den universellen Einsatz bietet PASCAL

Das Einlesen einer Datei



ausschließlich für Textdateien die Funktionen „EoLn (F)" und die beiden vordefinierten Prozeduren „ReadLn (F)" und „WriteLn (F)". Die EoF-(F)-Funktion kann natürlich bei allen Dateitypen verwandt werden. Da Zeilen-Endmarkierungen jede Form annehmen können, ergibt das Lesen von „char" bei EoLn=true ein Leerzeichen. Normalerweise wird daher vor dem Lesen auf EoLn überprüft.

Dateien eröffnen

Da „input" und „output" vor und nach jeder Programmausführung existieren, sind sie permanent eröffnet und brauchen nicht erst lokalisiert oder aufgebaut zu werden. Alle anderen Dateien müssen jedoch mit einem externen Systemnamen angesprochen werden und eröffnet sein, wenn mit der Prozedur „reset" daraus gelesen werden soll. Für das Schreiben einer Datei mit der Prozedur „rewrite (Datei)" gelten die gleichen Regeln. Der generelle Ablauf für die Bearbeitung einer Textdatei sieht folgendermaßen aus:

```
(* Datei eröffnen *)
WHILE (* nicht Ende der Datei *) DO
  WHILE (* nicht Zeilenende *) DO
    (* Zeichen lesen *)
    (* Zeichen verarbeiten *)
    (* Zeilenende überspringen *)
```

Die Prozeduren „read" und „write" arbeiten mit Dateibuffern und den einfachen Ein- und Ausgabebefehlen „put" (Ausgabe) und „get" (Eingabe von einer Datei). Beim Aufruf der Prozedur „rewrite" befindet sich so lange keine Information im Datenbuffer, bis eine Schreiboperation ausgeführt wird:

```
F^ := Daten;
put ( F )
```

Ähnlich funktioniert die Anweisung „read (F, Daten)":

```
Daten := F^;
get ( F )
```

Die Ein- und Ausgabeanweisungen unserer Copy-Prozedur:

```
read ( Quelle, Zeichen );
write ( Ziel, Zeichen )
```

könnten daher auch ganz anders, nämlich folgendermaßen aussehen:

```
Ziel^ := Quelle^;
put ( Ziel );
get ( Quelle )
```

In diesem Fall benötigen wir die lokale char-Variable „Zeichen" nicht. Mit folgendem Ablauf läßt sich der Vorgang von ReadLn (F) besser verstehen:

```
WHILE NOT EoLn ( F ) DO
  get ( F ); (* Rest der Zeile ignorieren *)
get ( F ) (* und EoLn-Zeichen überspringen *)
```

Nach einem „ReadLn" enthält der Dateibuffer immer das erste Datenelement der nächsten Zeile. Wenn EoLn (F) „true" ist, kann dies ein Leerzeichen sein, bei EoF (F) = true ist es jedoch undefiniert. In diesem Fall ist nicht nur der Versuch, aus dieser Datei zu lesen, illegal, sondern auch jede andere Dateioperation, wie das Testen des Dateibuffers. Andere Dateien als input und output müssen daher sehr sorgfältig programmiert werden.

Wir wissen nun, wie die verschiedenen Endbedingungen begonnen werden und können daher eine Prozedur schreiben, die Leerzeichen überspringt.

```
PROCEDURE LeerzeichenUeberspringen ( VAR
  F : text );
  CONST
    Leerzeichen = ' ';
  VAR
    fertig : boolean;
  BEGIN
    fertig := EoF ( F );
    IF NOT fertig THEN
      fertig := input^ > Leerzeichen;
    WHILE NOT fertig DO
      BEGIN
```

PASCAL legt bei der Eröffnung der Datei F automatisch den Bufferbereich F^ an und liest dort das erste Zeichen der Datei ein. Ein read-Befehl ordnet dieses Zeichen einer Variablen zu und überträgt das nächste Zeichen der Datei in den Bufferbereich. Sind die ersten Zeichen von F beispielsweise „PENIBEL", dann wird bei Eröffnung der Datei das P in F^ übertragen. Der erste read-Befehl ordnet das Zeichen P einer Variablen zu und ersetzt es dann im Buffer F^ durch das E.

```
            get ( F );
            fertig := EoF ( F );
            IF NOT fertig THEN
              fertig := F^ > Leerzeichen
        END
    END; (* LeerzeichenUeberspringen *)
```

Beachten Sie, daß diese Routine in einer Textdatei nicht nur alle Leerzeichen überspringt, sondern auch die Zeichen der Zeilen-Endmarkierung. Sollen nur die Leerzeichen einer Zeile übersprungen werden, muß die Bedingung folgendermaßen geändert werden:

```
fertig := EoLn ( F ) OR ( F^ > Leerzeichen )
```

Der Ablauf für die Beseitigung aller „leeren" Zeichen sieht daher so aus:

```
REPEAT
  LeerzeichenUeberspringen ( F );
  IF NOT EoF ( F ) THEN
    TextGefunden := NOT EoLn ( F )
UNTIL TextGefunden OR EoF ( F )
```

Mit dieser letzten Änderung lassen sich leicht interaktive Programme schreiben, die Leereingaben ausschließen:

```
REPEAT
  write ( 'Daten eingeben :' );
  LeerzeichenUeberspringen ( input )
UNTIL NOT EoLn ( input )
```

Zwar kann diese Routine nicht die Eingabe von Steuerzeichen abfangen, die das Dateiende markieren, doch können wir dafür immer auf die vorherige Prozedur zurückgreifen.

Die Prozedur „assign" wird inzwischen fast als „Standarderweiterung" angesehen und findet – wie „open" und „seek" für Random-Access-Dateien – vermutlich schon bald Eingang in alle PASCAL-Versionen. Zwei oft verwandte Erweiterungen sind auch:

```
FUNCTION Fstat ( DateiName )
```

die den booleschen Wert „true" liefert, wenn eine Datei bereits besteht, und

```
PROCEDURE Rename ( AlterName,
  NeuerName )
```

ermöglicht die Umbenennung von Dateien.

PASCAL-Dialekte

PASCAL kennt keine Beschränkungen in der Zahl der Dateien, die gleichzeitig eröffnet sein können. Da Dateien jedoch über das Betriebssystem des Computers aktiviert werden, gibt es hier Abweichungen von der strikten Einheitlichkeit der Sprache. So existieren PASCAL-Versionen, die Dateien nicht unterstützen.

Auch die von Betriebssystemen vorgegebenen Namensstrukturen für Dateien können Probleme verursachen. PASCAL-Versionen von Großrechnern nehmen beispielsweise die ersten zehn Zeichen eines Dateinamens und durchsuchen damit den Speicher nach der zugehörigen Datei. Viele Systeme verfügen jedoch nicht über diese Möglichkeit. Beispiele dafür sind: A:CP/M-FIL.DAT, #4:APPLEFOR.MAT und O:P.BBCFILE. Die „Standarderweiterung" für den korrekten Aufbau eines PASCAL-Dateinamens ist die Prozedur „assign". Sie wird vor „reset" oder „rewrite" eingesetzt:

```
assign ( Dateizuordnung, DateiName )
```

Für unser Programm genügt daher: assign (Datei, D:DateiNam.dat).

Das Programm „CopyText"

Die Zuordnung „text" kennzeichnet einen in PASCAL häufig eingesetzten Datei-Typ. Die Deklaration einer Textdatei-Variablen geschieht in der Parameterliste des Programmkopfes. Das folgende Programm kopiert Textdateien. Damit es sich für eine Vielzahl von Anwendungen einsetzen läßt, haben wir den Kopiervorgang als Prozedur angelegt.

Bedenken sie, daß Compiler, die nicht standardmäßig ausgelegt sind, die Anweisung „reset (F1,'Quelle')" im Hauptprogramm benötigen. Beachten Sie weiterhin, daß beide Dateien als VAR-Parameter an die Prozedur Copy übergeben werden und damit nicht nur die Zieldatei aktualisiert, sondern auch die Quelldatei gelesen und vom Status her verändert wird. Dateivariablen werden immer als Adreß-Parameter und nie als Wertparameter übergeben, da die letzteren immer lokale Variablen sind. Bei Dateien, deren Größe den verfügbaren Arbeitsspeicher überschreitet, muß ein Einlesen mit Wertvariablen daher immer zum Absturz des Programms führen. Aus Gründen der Platzersparnis wird hier jedoch für umfangreiche Strukturen vom Typ Array und Record oft eine Ausnahme gemacht.

```
PROGRAM CopyText ( F1, F2 );

VAR
  F1,
  F2 : text;

  (* ------------------------------ *)
PROCEDURE Copy ( VAR Quelle,
                     Ziel   : text );
VAR
    Zeichen : char;
BEGIN
  WHILE NOT EoF ( Quelle ) DO
    BEGIN (* eine Zeile kopieren *)
      WHILE NOT EoLn ( Quelle ) DO
        BEGIN
          read ( Quelle, Zeichen );
          write ( Ziel, Zeichen )
        END;
      (* jetzt das Zeilenende kopieren: *)
      ReadLn ( Quelle );
      WriteLn ( Ziel )
    END
END; (* Copy *)
  (* ------------------------------ *)

BEGIN (* CopyText — Hauptprogramm *)

  assign ( F1, 'Quelle' );
  reset ( F1 ); (* Datei finden und zum Lesen eroeffnen *)
  assign ( F2, 'Ziel' );
  rewrite ( F2 ); (* Datei anlegen *)
  Copy ( F1, F2 )

END.
```

Dynamische Daten

In dieser vorletzten Folge unserer PASCAL-Serie wird untersucht, wie sich mit Pointern, dem Heap und verketteten Strukturen Datenmengen von unbegrenzter Größe verarbeiten lassen.

Mit zwei Standardprozeduren (*new* und *dispose*) und einem speziellen Datentyp gewinnt PASCAL eine außerordentliche Flexibilität beim Einsatz dynamischer Speicherplatzverwaltung. „Pointer" sind Variablen, die statt Datenwerten (beispielsweise Ganzzahlen) „Zeiger" auf strukturierte oder unstrukturierte Datenobjekte enthalten. Zeiger werden wie Dateibuffer geschrieben (die auf den nächsten Datensatz der Datei zeigen, jedoch auf völlig andere Weise) und durch einen Aufwärtspfeil definiert. Dieser Pfeil muß vor einem Typennamen stehen, der dem Datentyp entspricht, auf den der Pointer zeigen soll:

```
TYPE
  GrossArray = ARRAY 1 . . 100 OF real;
  indirekt   = ↑ GrossArray;
```

ISO-PASCAL-Compiler verwenden dafür auch das Symbol (' @ '). Die Typendefinition reserviert eine Speicherstelle. Die Adresse eines großen Arrays wird dort allerdings erst gespeichert, wenn es mit der Prozedur *new* eingerichtet wurde. Bis zu diesem Zeitpunkt ist der Wert des Pointers – wie bei jeder anderen Variablen – undefiniert.

```
VAR
  Adresse : indirekt;
  Zahl    : integer;
```

reserviert den Speicherplatz für eine Maschinenadresse (16 Bits bei einem Acht-Bit-Rechner) und eine Ganzzahl. Beide Variablen wurden jedoch noch nicht auf einen bestimmten Wert initialisiert.

So wie wir Ganzzahlen vor dem Einsatz mit Null initialisieren, können wir auch Pointern den Wert *NIL* zuweisen. Das reservierte Wort NIL zeigt an, daß der Pointer gegenwärtig nicht benutzt wird. Sein Wert entspricht dem numerischen Wert Null. Die Variablen unseres Beispiels könnten daher folgendermaßen initialisiert werden:

```
Adresse := NIL;
Zahl    := 0;
```

Da (die Konstante) NIL eine Typenbezeichnung ist, wäre es eigentlich logisch, sie in einer Programmiersprache mit einem separaten Namen zu belegen.

Soll eine Pointer-Variable auf ein neues Datenelement zeigen, braucht nur die PASCAL-Prozedur *new* aufgerufen zu werden, die im Speicher dann den entsprechenden Platz reserviert und dessen Adresse in den Pointer schreibt. Wenn wir uns nun auf das Datenelement und nicht auf die Pointer-Adresse beziehen wollen, müssen wir den Pointer p „de-referenzieren" bzw. als p↑ schreiben. Beachten Sie, daß der Aufwärtspfeil nun wie bei dem Dateibuffer *hinter* dem Pointer-Namen steht und als „das Element, auf das p zeigt" bezeichnet wird.

Löschen des Pointers

Wenn Pointer-Daten, die mit *new* angelegt wurden, nicht mehr gebraucht werden, kann man die PASCAL-Prozedur *dispose* aufrufen, die das genaue Gegenteil von *new* bewirkt. Sie gibt den reservierten Speicherplatz wieder frei und weist dem Zeiger den Wert NIL zu. Das folgende Programm verdeutlicht diese Vorgänge. Aus Gründen der Einfachheit verwenden wir hier nur Ganzzahlen.

```
PROGRAM ZweiPlusZwei (output);
TYPE
  pointer = ↑integer;
VAR
  p1,
  p2        : pointer;
  Ergebnis : integer;
BEGIN
  new ( p1 );
  p1↑ := 2;
  new ( p2 );
  p2↑ := p1↑;
  Ergebnis := p1↑ + p2↑;
  WriteLn ( p1↑,'+',p2↑,'=',Ergebnis );
  dispose ( p1 );
  dispose ( p2 );
END.
```

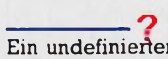
Diese etwas seltsame Art, zwei und zwei zu addieren, verdeutlicht zwei wichtige Punkte:

- Dynamische Variablen sind anonym – sie haben keinen Variablennamen (wie N), der irgendwo im Programm den Wert 2 enthält. Der Bezug auf die Datenelemente ist indirekt.
- Nach dem zweiten *dispose* existiert im Speicher nur noch die Ganzzahl *(Ergebnis)*; die dynamischen Daten gibt es nicht mehr.

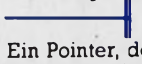
Wenn Sie unsere Serie über die Assemblersprache verfolgt haben, werden Sie sicher bemerkt haben, daß der indirekte Bezug der Pointer der indirekten Adressierung auf Maschinenebene entspricht. Zwischen diesen beiden indirekten Bezügen gibt es jedoch große Unterschiede. Zunächst sind in PASCAL nie die echten Adressen bekannt. Die einzige „absolute Adresse", die der PASCAL-Programmierer kennt, heißt NIL.

Auch können Sie in PASCAL den Speicher-

Pointer-Symbole



Ein undefinierter Pointer (entweder nach seiner Deklaration und vor einer Zuweisung, oder nach dispose).



Ein Pointer, der „nirgendwohin" zeigt (nach NIL-Zuweisung).

Eine Record-Einheit mit Datenfeld und Pointer.

platz beliebig oft belegen, freigeben und wiederverwenden, ohne je den Speicher „aufräumen" zu müssen. Da PASCAL die gesamte Speicherverwaltung übernimmt, interessiert nur, wieviel Speicherplatz noch frei ist. Diese Information läßt sich über die nicht standardmäßig vorgesehene Funktion *MemAvail* erfahren, – in einigen Versionen auch über die Funktion *Free*. Sie liefert die Anzahl der freien Speicherbytes. Auch die Funktion *SizeOf (Typenname)* ist sehr nützlich, da sie die Bytezahl des gewünschten Typs angibt.

PASCAL teilt den verfügbaren Arbeitsspeicher in zwei interne Strukturen – den Stack und den „Heap". Der Stack speichert beim Aufruf von Prozeduren und Funktionen die lokalen Daten, Rücksprungadressen und Ergebniswerte. Die dynamischen Daten werden dagegen dem Heap zugeordnet. Er arbeitet ähnlich wie der Stack, ist aber keine LIFO-Struktur (Last In, First Out). Der Heap fängt am anderen Ende des verfügbaren Arbeitsspeichers an und wächst in Richtung auf den Stack. Bei übermäßigem Gebrauch von *new* und/oder bei recursiven Programmen können Stack und Heap durchaus „zusammenstoßen". Dies läßt sich jedoch durch folgende Abfrage vermeiden:

```
IF SizeOf (Objekt) > Prozent * MemAvail
   THEN . .
```

Objekt ist der Typenname der Daten, die im Heap untergebracht werden sollen, und *Prozent* ein Wert im Bereich von 0,7, der dem Stack 30 Prozent des Speichers zur Verfügung stellt und dem Heap 70 Prozent. Wenn *MemAvail. Free* als „Hochwassermarkierung" eingesetzt wird, sollte auch eine Liste der nicht benötigten Pointer-Objekte angelegt werden.

Die eigentliche Stärke der Pointers zeigt sich erst beim Aufbau verketteter Strukturen. Für die String-Verarbeitung werden oft Arrays mit festen Längen (beispielsweise 80 Zeichen) eingesetzt. Ein in einem derartigen Array gespeichertes Schriftstück belegt jedoch auch für jede Leerzeile 80 Bytes.

PASCAL kann Schriftstücke mit variierenden Längen weitaus intelligenter speichern. Da PASCAL-Compiler zwischen Namen unterschiedlicher Länge unterscheiden können, ist hier die einfachste Struktur ein Record mit zwei Feldern: ein Feld für das Datenelement (in diesem Fall chars) und ein Pointer-Feld, das auf den nächsten Record der Liste zeigt.

```
TYPE
   String   = ↑Zeichen;
   Zeichen = RECORD
                ch        : char;
                naechster : String
             END;
VAR
   Zeile : String;
BEGIN
   Zeile := NIL; etc.
```

Ein String mit der Länge Null wird durch die Zuordnung des Wertes NIL dargestellt. Jede andere Zeichenfolge erhält für die einzelnen Zeichen (char) einen neuen Record und einen *Pointer* auf den darauffolgenden Record. Da der Pointer des allerletzten Records auf NIL gesetzt wurde, läßt sich auch das Ende des Strings leicht feststellen. Mit dieser Prozedur lassen sich die Strings ausgeben:

```
WHILE Zeile <> NIL DO
   BEGIN
      write (Zeile↑.ch);
      Zeile := Zeile↑.naechster
   END
```

Beachten Sie, daß die Datenstruktur recursiv ist (sie definiert sich selbst). Da dem Pointerfeld kein vollständig definierter Typ zugeordnet werden kann, ist hier der Bezug auf den nächsten Record möglich.

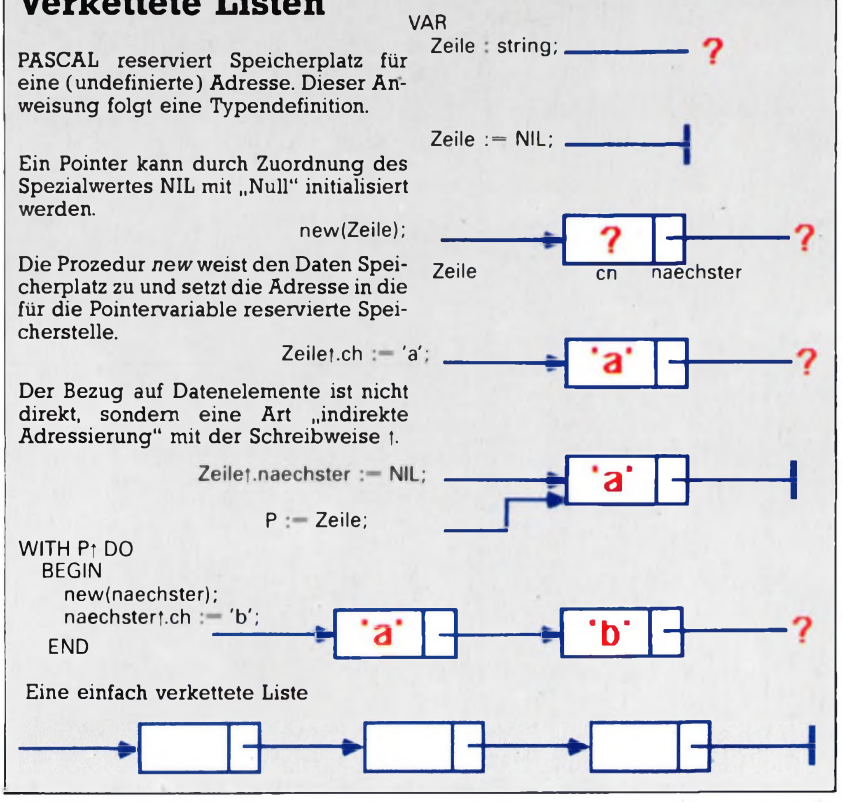
Verkettete Listen

PASCAL reserviert Speicherplatz für eine (undefinierte) Adresse. Dieser Anweisung folgt eine Typendefinition.

Ein Pointer kann durch Zuordnung des Spezialwertes NIL mit „Null" initialisiert werden.

Die Prozedur *new* weist den Daten Speicherplatz zu und setzt die Adresse in die für die Pointervariable reservierte Speicherstelle.

Der Bezug auf Datenelemente ist nicht direkt, sondern eine Art „indirekte Adressierung" mit der Schreibweise ↑.

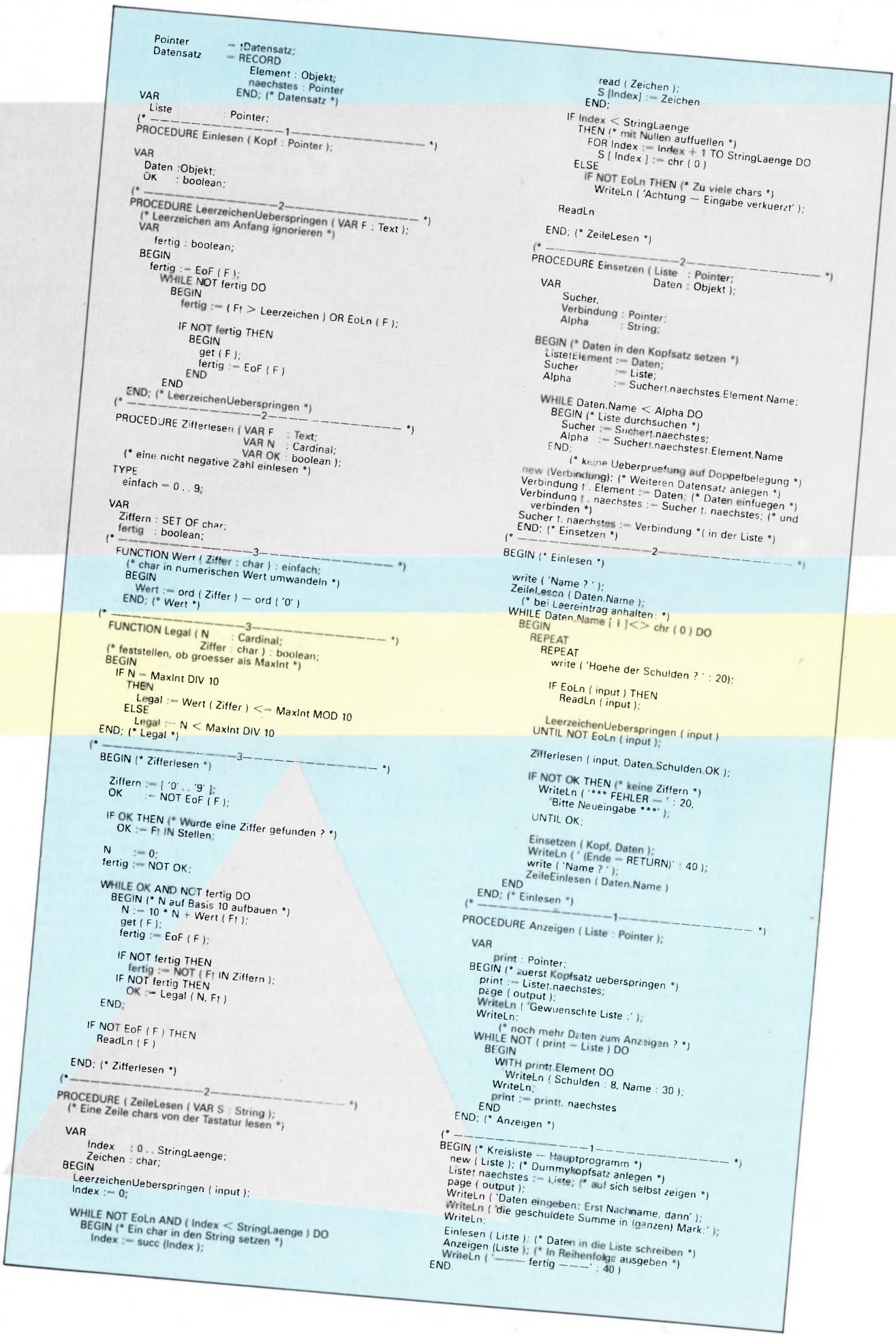


PASCAL bietet dem Programmierer die Möglichkeit, „verkettete Listen" einzusetzen. In diesen außerordentlich flexiblen Datenstrukturen zeigt jedes Datenelement auf das nächste Element der Liste. Die Listen können einfach verkettet sein (d. h. jedes Element zeigt auf das nächste) oder doppelt verkettet (d. h. die Pointer zeigen auf das nächste und das letzte Element der Liste) oder im Kreis verkettet (das letzte Element einer einfach verketteten Liste zeigt wieder auf das erste Element der Liste).

Das Programm Kreisliste

```
PROGRAM KreisListe ( input, output );
(* ------------------------------------------
 - Dieses Programm setzt Records mit vermischten
 - Daten in eine dynamische Kreisliste ein. Da
 - die Daten in alphabetischer Reihenfolge nach
 - dem Inhalt des Schlüsselfeldes (Name) einge-
 - setzt werden, sind keine zusätzlichen Sortier-
 - algorithmen notwendig.
 ------------------------------------------*)

CONST
   StringLaenge  = 25;
   Leerzeichen   = ' ';

TYPE
   Cardinal      = 0 . . MaxInt;
   StringGroesse = 1 . . StringLaenge;
   String        = PACKED ARRAY [ StringGroesse ]
                   OF char;
   Objekt        = RECORD
                      Name : String;
                      (* Platz fuer andere Felder *)
                      Schulden : Cardinal
                   END; (* Objekt *)
```

Mit diesem Programm können Sie Records mit gemischten Daten in eine dynamische Kreisliste einsetzen. Da die Daten nach einem alphabetisch geordneten Schlüsselfeld eingefügt werden, sind hier keine Sortieralgorithmen notwendig.



```
Pointer    = ^Datensatz;
Datensatz  = RECORD
               Element : Objekt;
               naechstes : Pointer
             END; (* Datensatz *)
VAR
  Liste          : Pointer;
(* ------------------1------------------ *)
PROCEDURE Einlesen ( Kopf : Pointer );

VAR
  Daten :Objekt;
  OK    : boolean;
(* ------------------2------------------ *)
PROCEDURE LeerzeichenUeberspringen ( VAR F : Text );
  (* Leerzeichen am Anfang ignorieren *)
  VAR
    fertig : boolean;
  BEGIN
    fertig := EoF ( F );
    WHILE NOT fertig DO
      BEGIN
        fertig := ( F^ > Leerzeichen ) OR EoLn ( F );

        IF NOT fertig THEN
          BEGIN
            get ( F );
            fertig := EoF ( F )
          END
      END
  END; (* LeerzeichenUebersprungen *)
(* ------------------2------------------ *)
PROCEDURE Zifferlesen ( VAR F   : Text;
                        VAR N   : Cardinal;
                        VAR OK  : boolean );
  (* eine nicht negative Zahl einlesen *)
TYPE
  einfach = 0 .. 9;

VAR
  Ziffern : SET OF char;
  fertig  : boolean;
(* ------------------3------------------ *)
  FUNCTION Wert ( Ziffer : char ) : einfach;
    (* char in numerischen Wert umwandeln *)
    BEGIN
      Wert := ord ( Ziffer ) - ord ( '0' )
    END; (* Wert *)
(* ------------------3------------------ *)
  FUNCTION Legal ( N      : Cardinal;
                   Ziffer : char ) : boolean;
  (* feststellen, ob groesser als MaxInt *)
  BEGIN
    IF N = MaxInt DIV 10
      THEN
        Legal := Wert ( Ziffer ) <= MaxInt MOD 10
      ELSE
        Legal := N < MaxInt DIV 10
  END; (* Legal *)
(* ------------------3------------------ *)
  BEGIN (* Zifferlesen *)

    Ziffern := [ '0' .. '9' ];
    OK      := NOT EoF ( F );

    IF OK THEN (* Wurde eine Ziffer gefunden ? *)
      OK := F^ IN Stellen;

    N     := 0;
    fertig := NOT OK;

    WHILE OK AND NOT fertig DO
      BEGIN (* N auf Basis 10 aufbauen *)
        N := 10 * N + Wert ( F^ );
        get ( F );
        fertig := EoF ( F );

        IF NOT fertig THEN
          fertig := NOT ( F^ IN Ziffern );
        IF NOT fertig THEN
          OK := Legal ( N, F^ )
      END;

    IF NOT EoF ( F ) THEN
      ReadLn ( F )

  END; (* Zifferlesen *)
(* ------------------2------------------ *)
PROCEDURE ( ZeileLesen ( VAR S : String );
  (* Eine Zeile chars von der Tastatur lesen *)
  VAR
    Index    : 0 .. StringLaenge;
    Zeichen : char;
  BEGIN
    LeerzeichenUebersprungen ( input );
    Index := 0;

    WHILE NOT EoLn AND ( Index < StringLaenge ) DO
      BEGIN (* Ein char in den String setzen *)
        Index := succ (Index );
        read ( Zeichen );
        S [Index] := Zeichen
      END;

    IF Index < StringLaenge
      THEN (* mit Nullen auffuellen *)
        FOR Index := Index + 1 TO StringLaenge DO
          S [ Index ] := chr ( 0 )
      ELSE
        IF NOT EoLn THEN (* Zu viele chars *)
          WriteLn ( 'Achtung — Eingabe verkuerzt' );

    ReadLn

  END; (* ZeileLesen *)
(* ------------------2------------------ *)
PROCEDURE Einsetzen ( Liste  : Pointer;
                      Daten : Objekt );
  VAR
    Sucher,
    Verbindung : Pointer;
    Alpha      : String;

  BEGIN (* Daten in den Kopfsatz setzen *)
    Liste^.Element := Daten;
    Sucher         := Liste;
    Alpha          := Sucher^.naechstes^.Element.Name;

    WHILE Daten.Name < Alpha DO
      BEGIN (* Liste durchsuchen *)
        Sucher := Sucher^.naechstes;
        Alpha  := Sucher^.naechstes^.Element.Name
      END;
          (* keine Ueberpruefung auf Doppelbelegung *)
    new (Verbindung); (* Weiteren Datensatz anlegen *)
    Verbindung^.Element := Daten; (* Daten einfuegen *)
    Verbindung^.naechstes := Sucher^.naechstes; (* und
      verbinden *)
    Sucher^.naechstes := Verbindung *( in der Liste *)
  END; (* Einsetzen *)
(* ------------------2------------------ *)
BEGIN (* Einlesen *)

  write ( 'Name ? ' );
  ZeileLesen ( Daten.Name );
    (* bei Leereintrag anhalten: *)
  WHILE Daten.Name [ 1 ]<> chr ( 0 ) DO
    BEGIN
      REPEAT
        REPEAT
          write ( 'Hoehe der Schulden ? ' : 20);

          IF EoLn ( input ) THEN
            ReadLn ( input );

        LeerzeichenUebersprungen ( input )
      UNTIL NOT EoLn ( input );

        Zifferlesen ( input, Daten.Schulden,OK );

      IF NOT OK THEN (* keine Ziffern *)
        WriteLn ( '*** FEHLER — ' : 20,
          'Bitte Neueingabe ***' );
      UNTIL OK;

        Einsetzen ( Kopf, Daten );
        WriteLn ( ' (Ende = RETURN)' : 40 );
        write ( 'Name ? ' );
        ZeileEinlesen ( Daten.Name )
      END
  END; (* Einlesen *)
(* ------------------1------------------ *)
PROCEDURE Anzeigen ( Liste : Pointer );
  VAR
      print : Pointer;
  BEGIN (* zuerst Kopfsatz ueberspringen *)
    print := Liste^.naechstes;
    page ( output );
    WriteLn ( 'Gewuenschte Liste :' );
    WriteLn;
      (* noch mehr Daten zum Anzeigen ? *)
    WHILE NOT ( print = Liste ) DO
      BEGIN
        WITH print^.Element DO
          WriteLn ( Schulden : 8, Name : 30 );
        WriteLn;
        print := print^.naechstes
      END
  END; (* Anzeigen *)
(* ------------------1------------------ *)
BEGIN (* Kreisliste — Hauptprogramm *)
  new ( Liste ); (* Dummykopfsatz anlegen *)
  Liste^.naechstes := Liste; (* auf sich selbst zeigen *)
  page ( output );
  WriteLn ( 'Daten eingeben: Erst Nachname, dann' );
  WriteLn ( 'die geschuldete Summe in (ganzen) Mark:' );
  WriteLn;
  Einlesen ( Liste ); (* Daten in die Liste schreiben *)
  Anzeigen (Liste ); (* In Reihenfolge ausgeben *)
  WriteLn ( '——— fertig ———' : 40 )
END.
```

GOTO ENDE

Dieser Artikel ist der Abschluß unserer PASCAL-Serie. Wir gehen nochmals auf die Beschreibung der Datenstrukturen ein und berühren einige Aspekte der Sprache, die bisher nicht erwähnt wurden.

Niklaus Wirth, der Vater von PASCAL, gab einem seiner Bücher den Titel „Datenstrukturen + Algorithmen = Programme" und deutete damit an, wie wichtig korrekte Datenbeschreibungen für die Algorithmen sind, die diese Daten bearbeiten. Wenn wir beispielsweise eine Zeichenfolge in einem *char*-Array unterbringen, brauchen wir für deren Verarbeitung völlig andere Abläufe als beispielsweise bei verketteten Listen.

Nehmen wir als Beispiel die einfache Aufgabe, die Länge eines Strings festzustellen. Bedenken Sie, daß ungenutzte Array-Elemente mit dem ASCII-Zeichen für Null, *chr* (0) ausgefüllt sind beziehungsweise dynamische Listen mit dem Pointerwert NIL abschließen. Die Array-Version sieht nicht kompliziert aus:

```
FUNCTION Laenge (S : String): Cardinal;
  VAR
    N          : 0..StringLaenge;
    gefunden   : boolean;
  BEGIN
    N          := 0;
    gefunden   := false;
  REPEAT
    N          := N + 1;
    gefunden   := S [ N ] = chr(0)
  UNTIL gefunden OR (N=StringLaenge);
  IF gefunden
    THEN
```

Das BaumSort-Programm

Dieses Programm setzt einige Prozeduren des Programms „KreisListe" aus dem vorigen Artikel ein. Ein Datensatz besteht aus einem Namen, dem Schuldenbetrag und anderen Feldern, die – falls gewünscht – hinzugefügt werden können. Die Daten werden über die Tastatur eingegeben und in den „binären Baum" eingefügt.

Jedes Strukturelement des Baumes hat zwei Pointer, die es mit den Elementen „darüber" und „darunter" verbinden. Neue Daten lassen sich einfügen, indem sie mit dem Namensfeld jedes Strukturelementes verglichen werden und dann in das entsprechende Verbindungsfeld (NiedrigZweig und HochZweig) verzweigen. Sobald ein leeres Strukturelement gefunden ist (mit einem NIL-Wert in der Verzweigung), wird das neue Element eingesetzt. Die Daten werden dann mit einem einfachen recursiven Vorgang in die Datei geschrieben.

Unser Beispiel läßt sich als Grundlage für eine Datenbank verwenden, die Sie genau auf Ihre Zwecke abstimmen können.

Für einfache Anwendungen ist die Kreisliste vermutlich besser geeignet. Da es dort keine NIL-Pointer gibt, entfällt bei jedem Vergleich auch der doppelte Test. Die Anzahl der NILs des binären Baums entspricht der Zahl der Strukturelemente + 1.

```
PROGRAM BaumSort ( input, output, Datei );
CONST
  DateiName     = 'BaumDaten';
  StringLaenge  = 25;
TYPE
  Cardinal      = 0..MaxInt;
  StringGroesse = 1..StringLaenge;
  String        = PACKED ARRAY [ StringGroesse ]
                    OF char;
  Objekt        = RECORD
                    Name     : String;
                    (* andere Felder *)
                    Schulden : Cardinal
                  END; (* Objekt *)
  Dateiart      = FILE OF Objekt;
  Baum          = ↑Zweig;
                  (* Vorausblick auf . *)
  Zweig         = RECORD
                    Element   : Objekt;
                    NiedrigZweig,
                    HochZweig : Baum
                  END; (* Zweig *)
VAR
  Daten         : Objekt;
  Datei         : Dateiart;
  Basis         : Baum;
INCLUDE 'Utils.src' (** Quellendatei mit:
                       LeerzeichenUeberspringen,
                       Einlesen und ZeileLesen;
                       — siehe Programm KreisListe **)
(* ———————————————————————— *)
PROCEDURE BetragLesen ( VAR Betrag : Cardinal );
  (* feststellen, ob Cardinalwert legal ist *)
  VAR
    OK : boolean;
  BEGIN
    REPEAT
      REPEAT
        write ( 'Betrag ? ' : 20 );
        IF EoLN ( input ) THEN
          ReadLn ( input );
        LeerzeichenUeberspringen ( input )
        (* bei Zeilenende Neueingabe ermoeglichen *)
      UNTIL NOT EoLn ( input );
      Einlesen ( input, Betrag, OK );
      IF NOT OK THEN
        WriteLn ( '——— FEHLER ———' : 20,
                  ' Bitte Neueingabe' )
    UNTIL OK (* Betrag muss legal sein *)
  END; (* BetragLesen *)
(* ———————————————————————— *)
PROCEDURE Wachsen ( VAR      Blatt : Baum;
    (* an den Baum anhaengen *) Daten : Objekt );
  BEGIN
    new ( Blatt );
    WITH Blatt↑ DO
      BEGIN
        Element     := Daten;
        NiedrigZweig := NIL; (* noch keine weiteren Daten *)
        HochZweig   := NIL (* nicht mehr in diesem
          Zweig *)
      END
  END; (* Wachsen *)
```

```
        Laenge := N — 1
      ELSE
        Laenge := StringLaenge
  END, (* Laenge *)
```

Hier können jedoch Mißverständnisse und Fehler entstehen. Warum benötigen wir beispielsweise die lokale Boolesche Variable *gefunden?* Und warum wird *Laenge* der Wert N−1 zugewiesen? Dies sind kleine Probleme. Da kompliziertere Algorithmen jedoch ein größeres Fehler-Risiko mit sich bringen, würden Strukturen dieser Art unnötig zusätzliche Verwirrung stiften.

Der Einsatz von „Bit-Flags" (Boolescher Variablentyp) ist einer der ältesten Programmierkniffe. Oft werden diese Typen jedoch nicht als logischer Teil des Algorithmus eingesetzt, sondern nur zur Steuerung des Computers. In unserem Beispiel muß daher die lokale Variable N verwandt werden, nicht *Laenge, da Laenge* Funktionsname ist und keine Variable. Wenn *Laenge* auf der rechten Seite einer Anweisung erscheint, wie:

```
Laenge := Laenge + 1
```

wird das Funktionsmodul recursiv aufgerufen.

Stellen Sie diesen Ablauf jetzt einer *Laenge*-Funktion für verkettete Listen gegenüber. Bedenken Sie dabei, daß die Typendefinition von *String* bei dynamischen Darstellungen anders ist, da hier Strings von beliebiger Länge entstehen können und Datenbeschreibungen daher oft recursiv definiert werden müssen.

Viele einfache Beispiele für Recursionen lassen sich auch als iterative Algorithmen darstellen. Sehen wir uns dazu die *Laenge*-Funktion für einen recursiven Stringtyp an:

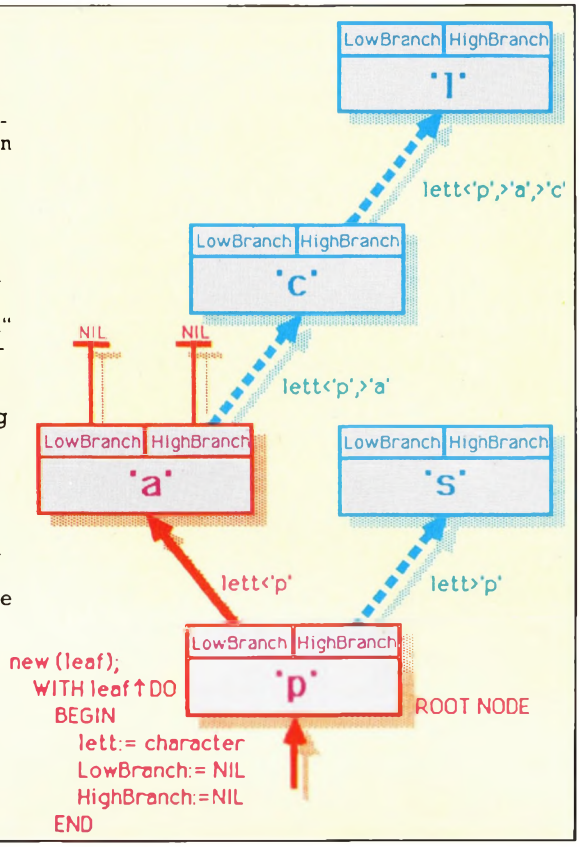
```
FUNCTION Laenge (S :String) : Cardinal;
  BEGIN
    IF S = NIL
      THEN
        Laenge :=0
      ELSE
        Laenge := succ(Laenge(S↑.next))
  END, (* Laenge *)
```

Am Anfang der Liste steht S, während *S↑.next* das Pointer-Feld des nächsten Datensatzes anspricht. Dieser Aufbau läßt sich auch als Liste ansehen, die mit dem nächsten Datensatz anfangt. Wenn wir kein NIL finden, wird die

```
(* ——————————————————————————— *)
PROCEDURE Klettern ( VAR Schoessling     Baum,
    (* Platz zum Einfuegen finden *) Daten   Objekt ),
  VAR
    Stamm     Baum,
    Groesser  boolean,
  BEGIN
    WHILE Schoessling <> NIL DO
      BEGIN
        Stamm    := Schoessling,
        Groesser := Daten.Name > Stamm↑
                      .Element.Name,
        IF Groesser
          THEN (* aufwaerts klettern *)
            Schoessling := Schoessling↑
              .HochZweig
          ELSE (* abwaerts klettern *)
            Schoessling := Schoessling↑
              .NiedrigZweig
      END,
    Wachsen ( Schoessling, Daten );
    IF Groesser
      THEN (* darueber einsetzen *)
        Stamm↑.HochZweig   := Schoessling
      ELSE (* darunter einsetzen *)
        Stamm↑.NiedrigZweig := Schoessling
  END, (* Klettern *)
  (* ——————————————————————————— *)
PROCEDURE Schreiben ( VAR     F          Dateiart,
    (* Daten in Datei schreiben *) Wurzel   Baum ),
  BEGIN
    IF Wurzel <> NIL THEN
      WITH Wurzel↑ DO (* Recursion *)
        BEGIN (* in Reihenfolge schreiben *)
          Schreiben ( F, NiedrigZweig ), (* zuerst
            niedrig, *)
          write ( F, Element ), (* dann die Wurzel *)
          Schreiben ( F, HochZweig ), (* dann hoch *)
        END
  END; (* Schreiben *)
  (* ——————————————————————————— *)
PROCEDURE Wiederholen ( VAR Wurzel   Baum ),
    (* Speicherplatz wieder verfuegbar machen *)
  BEGIN
    IF Wurzel <> NIL THEN (* Recursion *)
      BEGIN (* erst die Zweige — *)
        Wiederholen ( Wurzel↑.NiedrigZweig ),
        Wiederholen ( Wurzel↑.HochZweig );
        dispose ( Wurzel ) (* dann die Wurzel *)
      END
  END, (* Wiederholen *)
  (* ——————————————————————————— *)
BEGIN (* BaumSort — Hauptprogramm *)
  assign ( Datei, Dateiname ),
  page ( output ),
  WriteLn ( '=== BaumSort ===' : 25 );
  WriteLn;
  WriteLn ( 'Bitte Daten eingeben (erst Nach-
    name)' );
  write ('Name ? ' ),
  Einlesen ( Daten.Name ),
  BetragLesen ( Daten.Schulden ),
  Wachsen ( Basis, Daten ), (* Baumstamm
    'einpflanzen' *)
  write ('Name ? ' ),
  ZeileLesen ( Daten.Name ),
  WHILE Daten.Name [ 1 ] <> chr ( 0 ) DO
    BEGIN
      BetragLesen ( Daten.Schulden ),
      Klettern ( Basis, Daten );
      WriteLn ( 'RETURN = Ende'  40 ),
      write ('Name ? ' );
      ZeileLesen ( Daten.Name )
    END;
  (* Platz fuer weitere Bearbeitung, dann Datei
  schreiben : *)
  WriteLn ( 'Schreiben Datei . ', DateiName );
  rewrite ( Datei ),
  Schreiben ( Datei, Basis ),
  WriteLn;
  WriteLn ( 'Schulden' : 8, 'Name'  StringLaenge );
  WriteLn;
  reset ( Datei ), (* Datei lesen und in geordneter *)
                   (* Reihenfolge schreiben : *)
  WHILE NOT EoF ( Datei ) DO
    BEGIN (* Records einzeln lesen *)
      read ( Datei, Daten ),
          (* die Felder auf den Bildschirm bringen *)
      WITH Daten DO
        WriteLn ( Schulden  8, '  ', Name )
    END,
  Wiederholen ( Basis ); (* Speicher freisetzen *)
        (* fuer andere Aufgaben *)
END
```

Blatt für Blatt
Das Programm „Baum-Sort" setzt eine binäre Baumstruktur ein, um Daten nach Schlüsselfeldern geordnet speichern zu können. Die einzelnen Elemente werden dabei jeweils nach der alphabetischen Position des Namensfeldes dem hohen oder dem niedrigen Zweig zugeordnet. Die Prozedur „Wachsen" richtet neue Strukturelemente ein, während „Klettern" den Weg in den richtigen NIL-Zweig findet.

Das Diagramm zeigt das Einsortieren des pascal-Strings in eine einfache binäre Baumstruktur, deren Strukturelemente einzelne Zeichen (lett) enthalten. Die rote Markierung zeigt an, wie die Prozedur „Wachsen" den Baum vergrößert.



Laenge dieses Datenelementes bewertet und mit *succ* inkrementiert.

Viele Probleme lassen sich nur mit Recursionen lösen. Das vorstehende Beispiel kann jedoch auch ohne Recursion programmiert werden (wenn auch ein wenig umständlicher):

```
FUNCTION Laenge (S :String) : Cardinal;
  VAR
    N : Cardinal;
  BEGIN
    N := 0;
      WHILE S <> NIL DO
        BEGIN
          N := N + 1;
          S := S↑.next
        END;
    Laenge := N
  END; (* Laenge *)
```

Die Deutlichkeit dieses Algorithmus ergibt sich dabei wie selbstverständlich aus der recursiven Natur der Datenstruktur.

Viele PASCAL-Compiler unterstützen Compileranweisungen, die nicht Elemente der Programmiersprache sind. Der ISO-Standard erkennt dabei eigentlich nur die Anweisung *Forward*. Zwei Prozeduren oder Funktionen sind „wechselseitig recursiv", wenn sie sich gegenseitig aufrufen.

In dieser Situation wird zunächst der Kopf des einen Unterprogramms deklariert und sein „Inneres" durch die Compileranweisung *FORWARD* ersetzt. Nach der vollständigen Definition des zweiten Moduls wird der Kopf des ersten nochmals in abgekürzter Form (ohne Parameterliste) angegeben und dahinter der Inhalt des Moduls aufgeführt. Andere Compileranweisungen beeinflussen oft die Compilerzeit, doch sollte von dieser Möglichkeit nur wenig Gebrauch gemacht werden, wenn die Programme auch mit anderen Compilern funktionieren sollen. Viele Optionen sind nicht übertragbar und werden daher von anderen Compilern auch nicht ausgeführt.

Einige wenige Compiler (beispielsweise von HiSoft) haben eine andere Syntax für die Deklaration des Forward-Pointers. Hier kann nur das Handbuch weiterhelfen. Es gibt jedoch nur wenige PASCAL-Varianten, die nicht mit der ISO-Definition übereinstimmen.

In den Versionen UCSD, TCL/RML, Oxford und HiSoft wurde die Prozedur *dispose* durch die außerhalb des Standards liegenden Prozeduren *mark* und *release* ersetzt.

Eine weitere Datenstruktur von PASCAL muß noch erwähnt werden: die „Variante". Bisher haben wir für das Speichern unterschiedlicher Datentypen einen Record mit entsprechenden Feldern verwandt. Wie aber kann ein Teil oder der gesamte Record flexibel gestaltet werden, wie beispielsweise beim Speichern des Geburtsnamens einer verheirateten Frau? Zuerst wird der „feste" Teil des Records definiert und dann der Teil mit variierender Länge mit CASE und OR eingeleitet:

```
TYPE
  Art     = (maennlich, weiblich);
  Variant = RECORD
    (* erst die gemeinsamen (festen)
    Felder *)
    CASE verheiratet : boolean OF
      false : ();
      true : (Heiratsdatum : String;
              CASE Geschlecht : Art OF
                maennlich : ();
                weiblich  : (Geburtsname :
                             String))
    END; (* Variant *)
```

Beachten Sie, daß die leeren Felder mit Klammern gekennzeichnet sein müssen. Die längste Variante legt den Platzbedarf der Datei fest, doch läßt sich mit Pointern auf die Varianten – *new (p, true, maennlich)* eine Speicherplatzverschwendung verhindern.

Die wenigen Schwächen von PASCAL wurden im Laufe der Zeit behoben. Die Sprache ist außerordentlich flexibel, handlich, leistungsstark, für zahlreiche Aufgaben geeignet und leicht zu erlernen. Wenn Sie innerhalb des ISO-Standards bleiben, gibt es natürlich einige Abläufe auf Systemebene, die man nicht einsetzen kann. Diese Routinen lassen sich jedoch in Assembler oder BCPL schreiben und an das gewünschte PASCAL-Programm hängen. Der größte Vorteil von PASCAL ist jedoch die Möglichkeit, den entwickelten Code auf vielen Micros, Minis oder Großanlagen einsetzen zu können. PASCAL kommt damit einem Esperanto der Computerwelt nahe.